Interface

2003 October

［表紙デザイン：㈱プランニング・ロケッツ］

Contents 2003 October

特集

RISCプロセッサの基礎から最新プロセッサのしくみまで

Interface

Contents 2003 October

話題のテクノロジ解説

ショウレポート&コラム

一般解説&連載

情報のページ

連載「XScaleプロセッサ徹底活用研究」「フリーソフトウェア徹底活用講座」，「プログラミングの要」，「音楽配信技術の最新動向」は，お休みさせていただきます．

組み込み技術の総合展示会

第6回 組込みシステム開発技術展 ESEC

北村俊之

組み込みシステムの応用技術が一堂に会する展示会「ESEC」が7月9日(水)～11日(金)の3日間，東京ビッグサイトで開催された．主催はリードエグジビションジャパン(株)である．今回で第6回目を迎える本展示会は，国内外の主要企業300社が出展し，昨年を100社近く上回る過去最大規模での開催となった．また，NEC，東芝，ルネサス テクノロジ，富士通，インテル，アーム，AMDなど内外の主要な半導体ベンダの出展も話題の一つだった．

今回は展示会場全体に「組込みLinux」，「ユビキタス・ネットワーク」，「ボード・コンピュータ」，「設計・開発サービス/コンサルティング」の四つのゾーンが設置され，より明確な目的意識をもった来場者への便宜が図られていた．とくに今回新たに設けられた「ユビキタス・ネットワーク」は，ユビキタス環境に対応するソフトウェアからハードウェアまでの開発プラットホームを一堂に集め，技術者，開発者の高い関心を集めていたゾーンである．

また，「第12回 ソフトウェア開発環境展(SODEC)」，「第8回 データウェアハウス＆CRM EXPO」，「第5回 データストレージEXPO」も同時開催され，展示会全体としても出展企業600社以上と，過去最大規模での開催となっていた(**写真1**)．

〔写真1〕入り口のようす

● 組込みシステム開発技術

アイティアクセスでは，組み込み向けミドルウェアの各製品の展示デモを行っていた．暗号製品「C4シリーズ」，Webブラウザ「NetFront v3.0」などの定番製品をはじめ，ビデオ/メッセージコミュニケータ「Eyeball」なども来場者の関心の高いソリューションであるという．また，モトローラDragonBall MX1を搭載した「Linux開発リファレンスボード」も参考出品されていた．こちらは，MontaVista LinuxとGUI開発ツール，およびメトロワークスのPowerPartsが移植されているという．Windows CE.NETを搭載したDragonBall i.MX1リファレンスプラットホーム「SkyRider」の展示も行われていた(**写真2**)．

〔写真2〕アイティアクセスのSkyRider

ルネサス テクノロジは，同社のマイコンをベースとした自動車，ネットワークなどの数多くのソリューションと各種開発環境の展示，デモを行っていた．車載用のネットワークソリューションとして「SAFE-by-WIREデモセット」(**写真3**)の展示が行われており，今後自動車に20箇所以上搭載される，エアバッグの一括制御を行うためのソリューションとのことだった．

〔写真3〕ルネサスのSAFE-by-WIREデモセット

アームでは，最新の「RealView」開発ツールのデモとして，Multi-coreデバッグやOS Awarenessなどを行っていた．また，先日発表されたばかりの「RealView Developer Kit for OKI」の展示も行われていた．また同社のテクノロジを搭載した製品も数多く出展されており，中でもカネスタの「カネスタキーボード」(**写真4**)は来場者の注目を集めていた．同製品を組み込んだモバイル端末を机上に置き，投影された赤色キーボードパターン上でタイピングを行うと，センサが指の動きを読み取って文字入力を行うことができる．

〔写真4〕カネスタキーボード

● 「組込みLinux」ゾーン

モンタビスタソフトウエアジャパンは，コンシューマ機器向けの新製品である「MontaVista Linux Consumer Electronics Edition」を始めとする同社の主力製品と，これらを搭載したホームAVサーバ(NEC)，電話付Web端末(Panasonic)，チャネルサーバ(SONY)などが数多く展示されていた．民生家電機器への組み込みLinuxの対応が本格化していることが感じられた．また，パートナーのイーエルティによる組み込みLinuxの評価環境「Embedded Linux Riference Kit」の展示と共通プラットホーム戦略の紹介も行われていた．リネオソリューションズでは，同社の組み込みLinux「uLinux」と開発環境「ELITE」を中心とした展示を行っていた．

日進ソフトウェアでは，小型カメラとセンサ入出力を組み合わせるなどの，組み込みLinux搭載のフィールド用サーバの事例をデモを交えて紹介していた(**写真5**)．また，VisualStudio上でリアルタイムアプリケーションの開発を可能にする開発環境を展示するなど，来場者の関心を集めていた．

〔写真5〕日進ソフトウェアの小型カメラとLinuxサーバ

● 「ユビキタス・ネットワーク」ゾーン

ブライセンは，Bluetoothプロトコルスタック＆SDK，組み込みデータベース「Linter」や組み込みLinux「ELinOS」など同社製品を使用したシステムのデモを多数展示していた．オープンインターフェースでは，組み込み向けBluetoothプロトコルスタック「BULEmagic3.0」を始め，音楽データから感情を解析する「feel pop box」，IrMCテストツールなどのデモ紹介を行っていた．日新システムズでは，IP電話/VoIPに必要なQoS技術を備えた「TimeSysLinux」を始め，RJ45コネクタサイズの基板実装タイプデバイスサーバ「XPort」やLinuxサーバをリモートメンテナンス可能なコンソールサーバなど，多彩なソリューションで来場者の関心を集めていた．

● 「ボード・コンピュータ」ゾーン

アットマークテクノは，小型で低消費電力を実現したLinux対応のARM7 CPUボード「Armadillo」を中心に，展示を行っていた．また，応用事例として，このCPUボードとBluetoothを組み合わせたソリューション(**写真6**)が展示されており，来場者の注目度も高いとのことであった．ロムウィン(**写真7**)では，同社のROM-Winシリーズに対応した，ファンレス対応のパネルPCシリーズやロムターム II Aシリーズ，オールインワンCPUボードの展示が行われていた．同社のソリューションを利用すれば，システムを簡単にROM化でき，耐環境性に強いシステム構築が可能であることをアピールしていた．

〔写真6〕アットマークテクノのArmadillo

〔写真7〕ロムウィンのブース

Show report

組込みシステム技術に関するワークショップSWEST5

■日時：2003年7月23日（水）～24日（木）
■場所：遠鉄ホテルエンパイア（静岡県浜松市）

懇親会のようす

「第5回 組込みシステム技術に関するワークショップ（5th Summer Workshop on Embedded System Technologies）」，略称SWEST5が浜松湖畔で開催された．このワークショップは，一泊二日の日程で開催され，組み込み技術者による技術発表と交流を深めることを目的としたものだ．実行委員長の高田広章氏（名古屋大学）など，組み込み関連の著名人をはじめとした300人近い技術者が参加する大規模なイベントとなった．

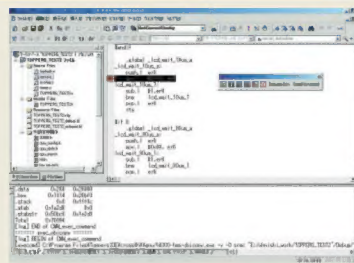
オープニングセッションでは，RoboCupの実行委員なども勤める松原仁氏（はこだて未来大学）による「災害救助からエンタテインメントまで――未来のITロボットの行方――」と題した講演が行われ，災害救助用ロボットの開発を目的とした「RoboCup-Rescue」の概要と救助技術の現状などが語られた．すでに9/11のテロでは，実際にロボットを用いたレスキュー活動が行われ，状況と用途を限定すれば現状でもロボットは役に立つとのことだった．

また，災害用システムは，「平時から役に立ち，使われるシステムであることが重要」という指摘がなされた．例として挙げられた災害救助支援システムは，平時には観光情報を，災害時には被災情報の提供を行うことにより，平時から役に立ち，なおかつ地理データなどが共用できるとの利点も挙げられた．

今回からはオープニングセッション以外の壇上での一般発表は取り止め，代わりにポスターセッションとして，各発表者が展示会場で個別に発表を行うという形式をとった．

（株）ヴィッツの「TOPPERS/JSPカーネル対応統合開発環境」は，Microsoft Visual C++上でTOPPERS用ソフトウェアを開発するための統合開発環境．従来のVisual C++のIDEからクロスコンパイル用のGCC/GDBなどを呼び出し，コンパイル/デバッグなどが行える．現在はH8/300Hでの動作確認を行っており，TOPPERSの対応している80%のCPUをカバーする予定とのことだ．

ヴィッツのTOPPERS/JSPカーネル対応統合開発環境

ほかの発表は「将来の組込みシステムへのOS仮想化技術の応用」（早稲田大学 追川修一氏/中島達夫氏），「ハッシュによるリクエスト判別の高速化を実現したUSB2.0デバイスコントローラの設計」（東海大学 名野　響氏ほか），「分散型組込制御システムのComponent BasedによるGUI開発ツール」（ユーシステムズ 上野真路氏ほか），「アクティブオブジェクトモデリングのこころ」（日本IBM 藤倉俊幸氏）など．また，（株）SRAによるQt/Embeddedの展示も行われ，同社が国内での販売・サポート業務などを行うことも発表された．

2日目はチュートリアルが行われた．「最近のプロセッサ・アーキテクチャ技術」（東京大学先端科学技術研究センター 中村宏氏）では，Power4やAlpha 21264などで使われているプロセッサ高速化技術のうち，分岐予測とスレッドレベル並列化技術について解説された．動的分岐予測方式として状態を2ビットで保持する方式は，「2回連続で予測が外れた場合のみ，予測を修正する」ことで，予測成功率を上げる手法である．

「オープンソースによる組込み開発と法的課題について」（イーエルティ 江端俊昭氏）は，GNU GPLをていねいに読み解き，解釈を加えるというもの．GPLは契約書であるにも関わらず著作権表示があるのは，改変を防止するためのものであるということ（著作物であれば，勝手に改変することができない），Linuxなどのローダブルモジュール形式のデバイスドライバがGPLかどうかは，さらに議論が必要になるが，デバイスドライバのほとんどの機能をBIOSで実現し，デバイスドライバ（GPL）からBIOS（非GPL）を呼び出すことでこの問題を回避することが可能，などの興味深い内容だった．

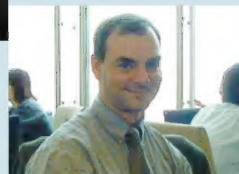
セッション，チュートリアルなどでの交流のほか，一泊二日での開催ということもあり，夜通し議論を行う参加者も多く，技術者の横のつながりを広げるイベントであった．

米I-Logix社がRhapsody製品群の日本における営業体制を強化

■日時：2003年7月11日（金）
■場所：ロイヤルパーク汐留タワー（東京都港区）

UML（Unified Modeling Language）に準拠し，MDA（Model Driven Architecture）をサポートしたグラフィカル開発ツール「Rhapsody」などを開発・販売している米I-Logix社（http://www.ilogix.com/）CEOのGene Robinson氏，Director, iNotion MarketingのJim McElroy氏が来日し，Rhapsody製品群の営業体制強化などを発表した．具体的な内容は，東芝情報システム（株）との販売提携，キャッツ（株）とのアライアンス，日本におけるサポートエンジニアの採用である．

I-Logix社CEOのGene Robinson氏

Robinson氏から日本の現場組み込み技術者へ向けたメッセージ――「私はいまの仕事の前にはEDA系の仕事をしていたが，あるきっかけで組み込み系の技術者が，厳しい開発サイクルや，コードサイズの制限などの厳しい情況にさらされていることに気付かされた．Rhapsodyを開発効率化のために役立ててほしい」

I-Logix社Director, iNotion MarketingのJim McElroy氏

アンソフト・ジャパン，HFSS V9.0発売

■日時：2003年7月7日（月）
■場所：アンソフト・ジャパン（神奈川県横浜市）

アンソフト・ジャパン（株）は，高周波3次元電磁界解析ツールHFSS V9.0を発売した．HFSS（High-Frequency Structure Simulator）は，米国Ansoft社が開発した高周波デバイス設計ツール．各種デバイスコンポーネントの動作原理であるマックスウェル電磁界方程式を数値シミュレーションにより解き，各種デバイス特性を算出するもの．価格は￥5,900,000～．

ハッカーの常識的見聞録

34

広畑由紀夫

今月の常識

「家庭でのInternetを快適にしよう!」

☆ 今回は，NTT東日本でも始まった「フレッツADSLモアⅡ」と家庭内LANの無線化について話していきます．

● ADSLの高速化

7月22日にサービスを開始したフレッツADSLモアⅡサービスへ，フレッツADSL8Mから契約を変更することになりました．筆者の環境では局から若干離れているため，事前情報からもあまり速度の向上は望めなかったのですが，より快適なインターネット接続環境を手に入れるべく，ADSLモデムやルータの取り替えなど家庭内LANの接続見直しを含めて，これを断行することにしました．

結果は，下り速度の向上こそ10%程度でしたが，上りは1Mbpsまで上がり，下りの速度向上がそれほどなかったにも関わらず，以前よりも体感速度が向上しました．さらに，上りでの複数のリアルタイムネットワークアプリケーションからの接続タイムアウトの確率も減り，複数のコンピュータおよびゲーム機器での平行プレイでの障害発生率が下がったようです．

● 無線LANについて

最近では802.11gの正式認可によって，802.11bでは物足りなかった無線LANの速度の向上と安定度の保証が進んでいます．筆者が必要とするマルチセッション接続対応機器では802.11g内蔵機器が現在まだ発売されていないため，別途アクセスポイントを立てたり，用途によっては802.11bを使用することなどで使い分けています．

802.11aに関しては，速度的には802.11gで代用できますし，WiFi認証も現在では与えられています．既存インフラが802.11aである場合は考慮する必要があるものの，新規もしくは802.11bの拡張であるならば，802.11gを考慮しておくだけで十分だと思われます．

● 認証設定

無線LANでは物理ケーブルを使わずに通信を行うため，暗号化なしや承認なしの接続は家庭での使用においても止めておきたいところです．「MACアドレスフィルタリング」，「暗号化設定」ぐらいは行っておきましょう．

暗号化は，ネットワーク内を流れるパケットを簡単には覗かれないようにしてくれます．完全に安全とはいえない部分も存在しますが，少なくとも非暗号化データを垂れ流すよりは安心できます．また，MACアドレスフィルタリング設定を行い，無線LANアクセスポイントへの接続を特定のネットワークカードからに限定します．これによって，他人によるネットワークの無断使用を防止し，無線LANから有線LANへ勝手にアクセスされることを防止します．

● 使用する無線チャネル

無線LANでは，使用するチャネルを設定します．このチャネルによって同一チャネル以外のアクセスを受け付けないようにし，エリアの重なる異なる無線LANでの通信競合を緩和します．

注意しておきたいのは，802.11g以前の802.11b専用で発売されていた一部の無線LANカードのドライバに不具合があるらしいことです．筆者が体験した不具合は「使用するチャネルに12～14チャネルを割り当てると，自動的に11チャネルとして働き，しかも11チャネルの無線LANアクセスポイントと正常に通信が開始できない」という現象です．

● 設定の手順

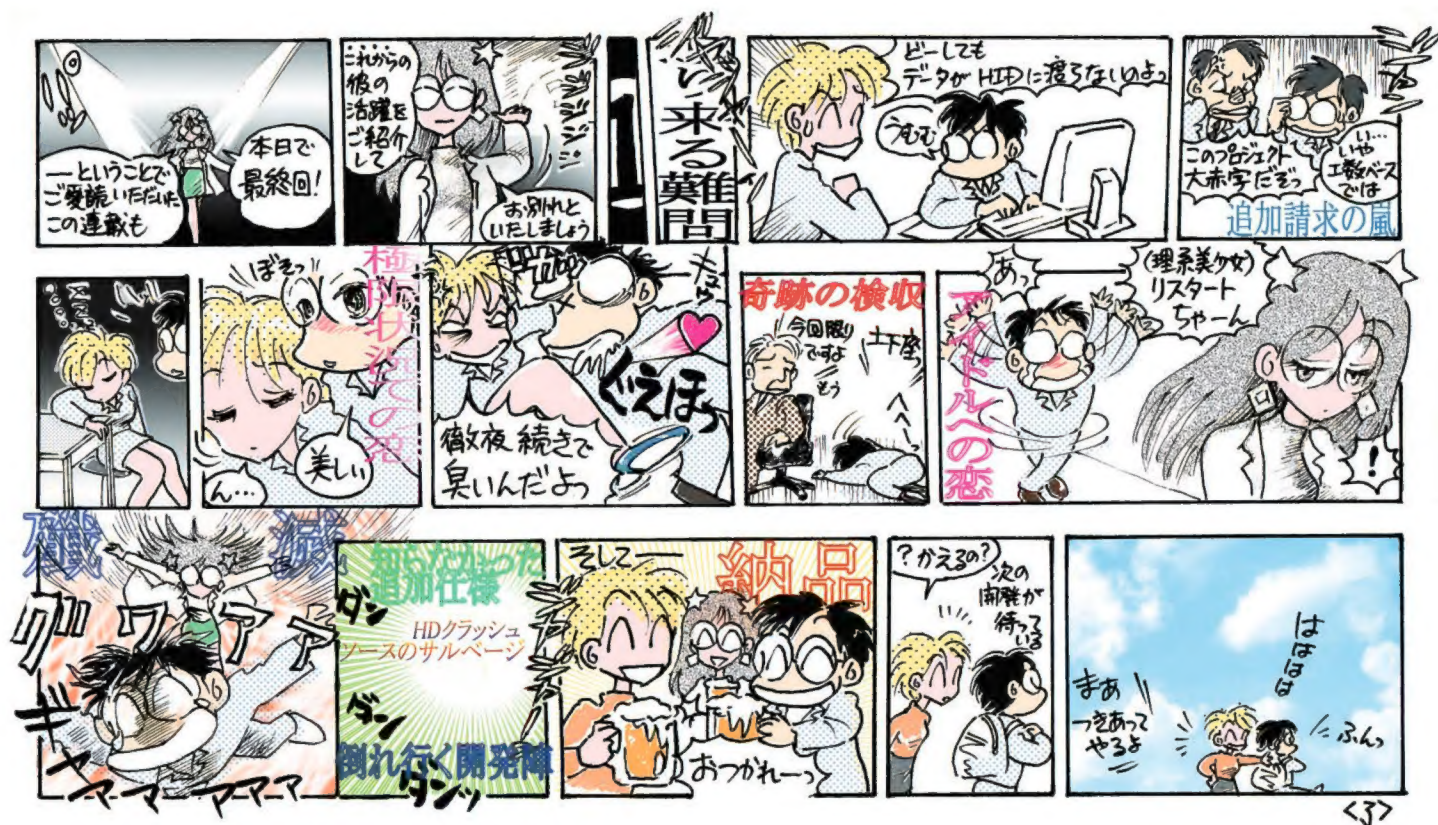
上記のような事柄から，導入設定のお勧め手順を考えてみましょう．手順としては，無線LANを設定し，それぞれにおいて無線LAN端末側から設定を反映したときにアクセス可能であるかどうかを確認していきます．とくに「WEP暗号化設定などを行った直後にアクセスできなくなった」というケースが多いため，WEP設定は最後にまわし，それ以前の接続を確認しつつ徐々にセキュアな設定にもっていくと失敗が少ないと思われます．

① まず使用するチャネルを自動のままで使用するか，使用するチャネルを設定する
② MACアドレスフィルタリングを使用し，アクセスするネットワークカードを指定する
③ SSIDを決めて，無線LANネットワークの名前を指定する
④ WEPの暗号化の有無，もしくは強度を設定する

● 無線LANアクセスポイント内蔵ルータを選ぶ

筆者が今回選んだのは，NTT東日本より発売されている無線LAN内蔵ルータ，FT6200Mです．そのため802.11g用のアクセスポイントを別途購入しましたが，複数の無線LANネットワークからのインターネット接続および有線LANの接続ともに利用できるようになりました．今後も高速な有線LANと，有線LANほどではないが比較的高速になってきた無線LANとの併用で，家庭内インターネット接続はさらに普及していくと思います．

ノートPCのバッテリ駆動時間も増えましたし，みなさんもケーブルから解放されたノートPCを屋外に持ち出しつつ，のんびりとネットワークを楽しんでみませんか？

ひろはた・ゆきお　OpenLab.

フジワラヒロタツの現場検証(72)〈最終回〉

現場検証，最後の挨拶

筆者のデイバッグには，ごちゃごちゃと物が入っています．古いPalm互換機．あと1年はもたせようと決意している1年半前の「最新」ノートパソコン．図書館で借りて，濡れないようにお手製カバーをつけた心理学の本．さまざまなPCMCIA規格のカード．携帯用に，整腸剤を入れたお菓子ケース（すぐお腹がくだるので）．PCケースを開けていじりまわす際になぜか手から血が出ていることが多いため，重宝するバンドエイド．もしものとき（？）に役立つソーイングキットなど，など．このデイバッグさえもっていれば，突然いつどこに行っても困らないように．

長く続けさせていただき，さまざまなエンジニアとしての「現場」にこだわってきたこの連載も，そろそろデイバッグをかついで立ち去る時期が来たようです．ここ10年ほど「ベンチャー」という横文字が付けられてきた中小零細企業のエンジニアとして，現場で思ったことや，感じてきたことを書いてきました．

連載中には，開発が修羅場の時期も多々ありました．「しょうもない」マンガとはいいながら，それなりに時間をとられるもので，筆者としては，どんなに忙しくても開発が最優先，そんなときには，泣きながらオチを考えておりました．

この間，筆者の仕事の内容も，社会と技術，自分の変化に呼応して移り変わってきました．さまざまな新しい技術に対応することよりも，組織の運営やプロジェクト管理，営業に時間を費やすことが多くなったのは，まったく，年齢のせいでしょうか．

けれど，そういった分野にも面白いものはいろいろあるもので，たとえば，見積もりのための方法論であるファンクションポイント（FP）法などは，大きなプロジェクトだけではなく，組み込みにも応用できるらしいということを聞きかじったりもしました．これはもっと詳しく調べてみようと思いながら果たせないままで，まったく自分の勉強不足には恥じ入るばかりです．

技術についても，オブジェクト指向でいうデザインパターンの本も読んでいないのに，より人間臭い，失敗についてのエピソード満載の，『アンチパターン』を面白がったりしていますし，最近，エクストリーム（XP）プログラミングという開発手法を知るにつけ，プログラミングの問題を解決するには，ニンゲン系をいじり回すことがカギになる，との感をますます強くもってきました．

当連載は結局，仕事が忙しいという愚痴と，技術をフォローアップするのがたいへんだ，という悲鳴に終始してきたような感がありますが，その中に，もしそういったニンゲン系の話をある程度語ることができたとすれば，読者の皆さんに支えられてのことと感謝しています．

シャーロックホームズに「最後の挨拶」という一編があり，ホームズがワトスンに「東風が来るね」というくだりがあります．受託開発の現場と，かつて「マイコン」を担ってきた世代にも，さまざまな厳しい風が吹いてきているような気がします．

本当に，思いのほか長く，この文章付きマンガ（マンガ付き文章？）を続けることができたのは，望外の幸せでありましたと，デイバッグを背負い直しつつ，お別れの挨拶といたします．

藤原弘達　（株）JFP　エンジニア，漫画家

特集 RISCプロセッサの基礎から最新プロセッサのしくみまで

詳説 マイクロプロセッサ ——パイプラインとスーパースカラ

特集執筆：中森 章

パソコンはいうに及ばず，情報家電から白物家電まで，現代社会はプロセッサなしには成り立たない．エレクトロニクス技術者として，プロセッサの理解は，必須項目といってよいだろう．そこで，今月号と来月号の2号にわたり，現在の最新プロセッサでも主流となっているRISCプロセッサに焦点を当て，徹底的に解説する．

RISCプロセッサの共通するアーキテクチャとしては，ロードストアアーキテクチャ，パイプラインを用いた1クロック/1命令実行，遅延分岐などがある．パイプラインの基本は1クロック/1命令だが，さらにプロセッサの高速化を推し進めるには，1クロックで複数命令を同時並列に実行するという方法が考えられる．これがスーパースカラである．

前編となる今月号では，このパイプライン処理とスーパースカラについて，もっとも基本的な構造のプロセッサから最新プロセッサまで，実際の各種アーキテクチャのプロセッサの事例を示しながら，徹底的に解説する．

Contents

Prologue

コンピュータの誕生からプロセッサの発展まで

マイクロプロセッサの歴史

中森　章

はじめに

そもそも**マイクロプロセッサ**(MPU)とは何なのだろうか．多くの人は，MPUがMicro Processing Unit(小型処理装置)の略語であり，コンピュータの中心的な動作を制御するLSIであることを知っている．その意味でCPU(Central Processing Unit：中央処理装置)と呼ばれることもある．本特集では一部を除きMPUと呼ぼう．

では，なぜMPUに8ビット，16ビットあるいは32ビットという種類があるのか，なぜ16ビットMPUよりも32ビットMPUのほうが処理性能が優れているのか，という点について知っている人は意外に少ないのではないだろうか？

これをひもとくには，まず大型計算機の歴史から振り返ってみる必要がある．歴史を探ることで，MPUの未来もおのずから見えてくるのではないだろうか．

● コンピュータ(計算機)という発想はいつから？

計算の機械化は古くから考案されてきた．16世紀にフランスの数学者のパスカル(Blaise Pascal)が考案したパスカリーヌ(Pascaline)をはじめとして，ドイツの数学者のライプニッツ(Gottfried Wilhelm Leibniz)がパスカリーヌを拡張した計算機を完成させている．

現在の感覚に近いコンピュータを最初に構想したのは，19世紀のイギリスの数学者であるバベッジ(Charles Babbage)といわれている．彼は1819年頃から，高信頼度の数表を階差法により作成する歯車式の階差機関の作成に着手した．1832年には試作機が作成されたが，政治的な理由で階差機関の開発は成功しなかった．その直後，階差機関の計算能力を上げる目的で，バベッジは解析機関を発案した．その複雑なメカニズムを実現するために，データ(記憶領域)と演算を分離することが考えられた．データから独立した演算器は，一連の指令(プログラム)を与えることで，種々の計算に対応できた．これがプログラム制御のはしりである．階差機関は階差法という計算に特定された専用マシンであったが，解析機関はある程度の汎用性をもっていた．これをもって解析機関を最初のコンピュータとみなす向きもあるが，プログラム内蔵方式ではなかったし，条件分岐機構をもっていないという意味では，コンピュータでないという意見もある．

その後，バベッジの業績は何人かの研究者に受け継がれたが，どれも試作程度で終わっている．バベッジ以後，1940年代までコンピュータ開発の表立った動きはなかったというのが定説である．この期間は100年の空白といわれている．

もっとも，これは米国中心の史観である．実際には，1930年代に機械式やリレー式がドイツのツーゼ(Konrad Zuse)らによって研究/試作され，またシュレイヤー(Schreyer)やアタナソフ(John V. Atanasoff)らが真空管方式によって開発を始めている．ツーゼのコンピュータは「Z1」，アタナソフのコンピュータは「ABC」として歴史に名を残している．

チューリング(Alan Turing)は1936年にチューリングマシンに関する論文を発表し，これが現代コンピュータの基礎理論となっている．チューリングもまた，第二次世界大戦中に暗号を解読するための「ボンベ」というチューリングマシンを応用した機械(コンピュータの原点)を開発している．ボンベはリレーを使用していたが，真空管を使用した電子計算機もチューリングの提案で開発された．これも暗号解読用である．

リレーは電磁石でスイッチをON/OFFするものだが，電気で同様の処理を行う真空管を使用すると1,000倍近い計算速度を得ることができる．これは1943年にCOLOSSUSという計算機として実現している．また，これはイギリスの話であるが，アメリカでも同時期に真空管を使用したENIAC(Electronic Numerical Integrator and Computer)という計算機が開発されていた．ENIACは実戦には間に合わなかったが，COLOSSUSは戦時中稼動していた唯一のコンピュータとして後世に名を残している．それは戦後も30年にわたって諜報活動に活用されるが，当時は機密事項として関係者が知るのみであった．

● アタナソフのコンピュータ ―― ABCマシン

1970年頃までの定説では，世界最初のコンピュータはモークリー(John W. Mauchly)とエッカート(J. Presper Eckert)およびゴールドスタイン(Herman H. Goldstine)によって1945年に開発されたENIACということになっていた．その説に一石を投じたのが，1937年からアタナソフとベリー(Clifford E. Berry)の開発したABC(Atanasoff-Berry-Computer)マシンである．これは，ガウスの消去法を想定した真空管式の計算機で，1942年にはほとんど完成していたといわれる．現在では，このABCマシンこそがENIACに先立つ世界最初のコンピュータといわれることが多い．

もともとアタナソフの業績は世間から忘れ去られていた．それを白日の下に引き戻したのは，1960年代に始まった，ENIACの基本特許に関する係争である．この裁判でENIAC特許が無効になったが，その根拠の一つとしてモークリーがアタナソフからENIACの基本原理を得ていたということが挙げられた．

事実，1941年にモークリーはアタナソフを訪問してABCマシンを見学していた．かくしてアタナソフの名前は一躍クローズアップされることになる．しかし，ABCマシンは29変数までの連立一次方程式の解法機にすぎず，プログラム内蔵という観点から見てもコンピュータと呼ぶにはふさわしくない．

● 真空管のコンピュータ

初期のコンピュータは，人間が手計算でやっていてはとても終了しないほど多量の計算を高速に行わせるために開発された．昔はそれほどの需要があったわけではないが，第二次世界大戦の頃になると大規模な計算の必要性が顕在化してきた．その主たる用途は軍事目的であったことは否めない．現在のコンピュータのはしりは1945年にペンシルバニア大学で作られたENIACといわれているが，これは大砲の弾道計算をするために作られたコンピュータである．その処理能力は現在のコンピュータと比べてはかわいそうなくらい低く，どちらかというとプログラム電卓といった感が強かったようだ．ENIACは，ある程度のプログラムを内蔵することもできたし，条件分岐機構ももっていたので，最初のコンピュータという栄誉を受ける資格は十分にある．ただし，これを主張する学者はCOLOSSUSの存在を無視しているようにも思える．ただ，COLOSSUSは暗号解読専用という観点から，汎用コンピュータとは認めてもらえないのだ．

1989年，米国のスミソニアン協会がアメリカ歴史博物館でコンピュータ開発の歴史展示を試みたとき，コンピュータの発明者はモークリーとエッカートになっていた．それが政治的圧力でうやむやな表現に変更された．その中で，アタナソフは最初のコンピュータを発明したが動作させることはできなかった，と説明されたという．

ENIACの本体は，30m × 90m × 3mの筐体の中に18,000本の真空管と10,000個のコンデンサを詰め込んだものである．このため，ENIACを設置するためにはまるまる一部屋分のスペースが必要だった．また，多くの真空管を動作させるために機関車並みの電力が必要だったという．それでも，電気式の真空管を使用するため，計算機の処理能力は機械式のリレーに比べて飛躍的に向上した．しかし，真空管を使っているために「図体がでかい」，「熱い」，「壊れやすい」というのが当時のコンピュータの常識だったようだ．この真空管の問題を何とかしない限り，コンピュータの発展はありえなかった．

● フォン・ノイマンに対する誤解

フォン・ノイマン（Jhon von Neumann）は，今日のコンピュータアーキテクチャの基礎を創造した人物として広く知れ渡っている．事実，プログラム内蔵を基本とする今日のコンピュータは「ノイマン型」といわれている．しかし，これはモークリーやエッカートの名誉を著しく傷つけるものである．

1944年の初め，ENIACの設計が始まってから18か月が過ぎた頃，フォン・ノイマンはゴールドスタインと会う機会を得た．そこで，フォン・ノイマンはゴールドスタインが関わっていた現在進行中のENIACの計画に非常に興味を覚えた．当時フォン・ノイマンは，原子爆弾を開発するマンハッタン計画の顧問をしていたが，この戦争に役立ちそうなコンピュータのことは知らされていなかった．これは，ENIACのスポンサーともいえる国防研究委員会（NDRC：National Defense Research Committee）がENIACを信用しておらず，取るに足りないものと考えていたからである．しかし，フォン・ノイマンは非常に興味を覚え，1944年の9月にENIACの開発現場を訪れ，ENIACの秘密情報へのアクセスが許された．

〔写真A〕ミシガン大学に展示されているENIAC（写真提供：近藤和彦氏）

さて，ENIACにはプログラミングが難しい，メモリが少量しかないという欠点があった．関係者の多くはENIACの完成のかなり前から，後継機種のEDVAC（Electronic Discrete Variable Automatic Computer）の議論を始めている．そして，1945年3月，フォン・ノイマン，モークリー，エッカート，ゴールドスタインらがEDVACの設計に関して議論した記録が，いわゆる「EDVACレポート」として，フォン・ノイマンの単独名で，機密事項であるにもかかわらず，世界のコンピュータ技術者の間に広く流布された．フォン・ノイマンがEDVACの設計に参加する前にプログラム内蔵方式は考案されていた．しかし，この文書により，フォン・ノイマンがプログラム内蔵方式のコンピュータの発明者として誤って伝わってしまったのである．フォン・ノイマンは発明者ではないが，プログラム内蔵方式を論理的に明確にして発展させた業績は認めるべきであろう．

EDVACは，関係者間の意見の対立により大幅に開発が遅れ，頓挫してしまう．一方，フォン・ノイマンはプリンストン大学で新しいコンピュータの開発を指導することになる．そんな中，世界最初のプログラム内蔵方式のコンピュータとしての栄誉を勝ち取ったのは，EDVACの影響下にイギリスのケンブリッジ大学でウィルクス（Maulice Wilkes）により製作され，1949年5月に稼動を始めたEDSAC（Electronic Delay Storage Automatic Calculator）である．この名称はEDVACを意識して付けられたという．なお，ウィルクスはマイクロプログラミングの提唱者

としても有名である．

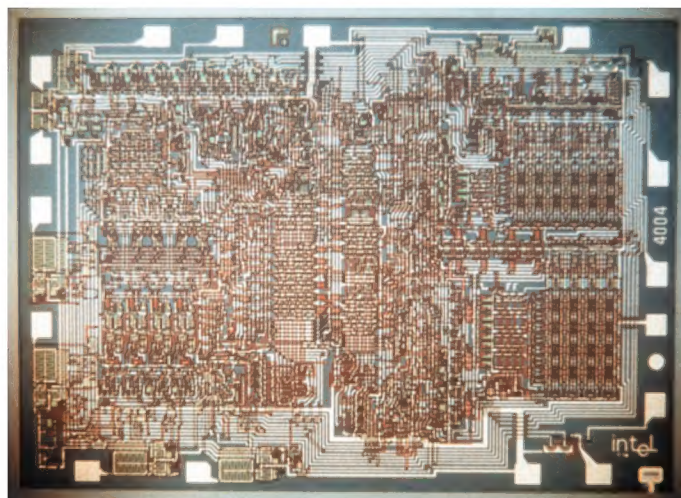
さて，これ以後も計算機の試作は星の数ほど行われ，IBMなどの大型計算機やスーパーコンピュータの開発へとつながっていくのだが，その歴史を追うことは筆者の意図ではない．今後はマイクロプロセッサの進歩を主として見ていこう．

● トランジスタが登場した！

コンピュータにとっての朗報は，1947年も終わりに近付いたクリスマスの2日前に訪れた．ベル研究所のウィリアム・ショックリー(William Shockley)，ジョン・バーディーン(John Bardeen)，ウォルター・ブラッテン(Walter Brattain)によってトランジスタが発明されたのだ．言うまでもなくトランジスタは，半導体産業において20世紀最大の発明である．トランジスタは真空管と違って熱をもたないし，壊れにくく，真空管より高速に動作する．そして，サイズが小さいのがなによりの利点だった．このトランジスタは，ラジオや補聴器など多くの電子機器の中心的デバイスとして確固たる地位を築いていくことになる．

当然，トランジスタを用いたコンピュータも作られた．FORTRANやCOBOLといった高級言語のコンパイラが登場したのは，トランジスタのコンピュータが全盛になる1950年代の後半から1960年代にかけてのことだった．この時期の代表的なコンピュータとして，IBMの7070や7090がある．まだ，マイクロプロセッサは誕生していない．

さて，トランジスタを用いてコンピュータを作る場合，最大の問題点は回路規模が大きく複雑であるということだった．数百ものトランジスタやコンデンサをハンダ付けしていく作業は人間の手によっていたが，それでいて十分な信頼性を得るのは至難の技だった．その障壁を乗り越えてコンピュータを作ったのだから，当時のコンピュータメーカーの頑張りには脱帽する．しかし，力まかせに作るコンピュータにはおのずと限界がある．人類には理論的には可能であっても，実装技術の未熟さゆえに到達できない夢がいくつもあったのだ．

〔写真B〕初のマイクロプロセッサ4004(写真提供：嶋正利氏)

● 集積化の時代

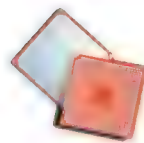
コンピュータにとって第2の転機は，1959年に訪れた．テキサス・インスツルメントのジャック・キルビー(Jack Killbey)とインテルの創始者の一人であるロバート・ノイス(Robert Noyce)が，シリコンウエハ上に抵抗やコンデンサを作るというアイデアを実現させたからだ．これがIC(集積回路)の誕生である．それまでは構成要素が独立した多くの部品であったため，それらを接続する困難さが生じる．それならば，いっそのこと，一つのチップに構成要素を作り込んでやれば接続の手間が省けるばかりでなく，非常に小型化できるというのがその基本的なアイデアである．もちろん，思い付きだけでICが製造できるわけではないが，とにかく，数々の製造上の困難を乗り越えた奇跡のチップとしてICが誕生した．

そして，ICが登場してから10年後の1969年，人類は月面上に小さいけれど人類にとっては大きな一歩を印すことができるようになった．あのアポロ計画である．しかし，合理的なアメリカ人が道楽(?!)で月まで行くわけはない．その背後に，宇宙の軍事利用という暗い影を宿していたことは厳然たる事実である．しかし，その技術が民間用に転換されてきて，われわれ一般人もその恩恵に預かることができたのは一筋の光明かもしれない．ICの初めての応用例は補聴器だったというし，ICがなければ現在のように信頼性の高いテレビ，ビデオ，DVDプレーヤなどのAV機器を手にすることもなかったはずである．

● マイクロプロセッサの鼓動

マイクロコンピュータのMPU，すなわちマイクロプロセッサの誕生には日本が大きく関わっている．なぜなら，マイクロプロセッサの物語は東京に端を発するからである．

1968年，日本の事務機器メーカーであるビジコン社は画期的なプリンタ付き電卓を作った．この電卓はプログラムをROMから読み出して実行するという，現在のコンピュータに近い形式を採用していた．しかも，この電卓はROMの内容を変更するだけでまったく別の電卓を作ることができるようになっていた．

1969年に入ると，電卓の高性能化，多様化，低価格化，高信頼化などの要請から，電卓をLSI化する計画が生まれた．しかし悲しいかな，日本には電卓程度の複雑さをもつ回路をLSI化する技術すらなかった．そこでビジコンは，インテルに援助を求めた．そのときインテルは，ビジコン側の示すLSIの規模が他社の電卓用LSIに比べて大きい(約16個の異なるチップを使用する)ので商売にならないと判断し，代わりに4ビットのMPUというアイデアを提示してきた(本音はチップ1個分の開発コストしかなかった)．これは，電卓のプログラムに使われていた命令をもっと低レベルの機械語レベルに引き下げて，汎用性をもったLSIをねらったものである．ビジコンはもともとプログラム方式の電卓を作っていた経験から，この新しいアイデアをすんなりと受け入れることができた．結果としてビジコンとインテルの折衷案で，世界初の4ビットMPUが作られることになったという．

電卓用のLSIに見られるように，当時のLSIの多くはカスタムデザイン（固有の目的のための設計）によって作られていた．特殊な目的をもったLSIを数多く短期間に製造していくためには，プログラム可能な汎用LSIというアプローチは非常に有用な解答だった．これがゆくゆくは「部品としてのコンピュータ」という市場を産んでいくことになる．

今ではインテルの主要製品となっているMPUであるが，当初インテルの幹部はその将来性に気付いてなかった．電卓の部品という認識で，メモリ事業が主体のインテルにとってはサイドビジネスの一環でしかなかった．MPUがインテルの未来をもたらす存在であることに気付くのは，発表から15年くらい経った後だという．

インテルが4004の所有権をもつようになるのは，積極的な理由からではなかった．4004の開発中に電卓事業が不振になり，ビジコン側が契約料の大幅値下げを要求してきた．インテルは，電卓市場以外での4004の外販権を無料で提供することを条件に，その要求を飲んだ．

かくして，1971年11月15日，インテルは4004を世界初のマイクロプロセッサとして発表した．これがマイクロプロセッサの誕生である．当時のインテルの最高経営責任者は，「人類史上でもっとも革命的な製品の一つ」とコメントしたらしい．しかし，多くの人々は4004のコンセプトを理解できず，インテルを脅威と思う者はほとんどいなかった．革新的なものの最初は，おしなべてこんなものかもしれない．

4004は750KHzのクロックで動作し，1命令の実行には最低8クロック必要だった．これは，1クロック実行が当然の，現在のRISC技術から見れば隔世の感がある．4004のクロックに関しては，インテルの公式資料では108KHzとなっているが，これは誤りである．4004のニュースリリースで命令の実行速度が10.8μsとなっていたのを勘違いしたものと思われる．この後に登場する8008の動作周波数も200KHzとなっている文献が多いが，500KHzの誤りである．

● マイクロプロセッサの展開

1972年4月1日，インテルは4004のアーキテクチャを拡張して8ビットデータ（文字データ）を扱えるようにした8ビットMPUの8008を発表した．動作周波数は500KHzだった（後に800KHz品も開発される）．8008はテキサスの端末メーカーであるデータポイント社からの受注である．しかし，データポイント社が契約料を払えなかったため，インテルは8008の命令セットの使用権とチップの外販権を獲得する．のちのx86命令セットの萌芽である．

8008は4004の後継と説明されることが多いが，4004の開発中にその技術者を引き抜いて開発したという．その意味で4004と同世代の兄弟チップである．

そして1974年，8008の改良版である，同じ8ビットMPUの8080が発表されるにあたって，マイクロプロセッサが本格的に市場に受け入れられるようになった．そして人々は，マイクロプロセッサによって多くの製品に知能を与えることができると考え，無数の新しい応用を夢に描いていった．

約55年前，部屋いっぱいの設置場所と機関車並みの電力を必要としたコンピュータが小指大（現在の規模では親指大というほうが適切か）のマイクロプロセッサへと凝縮されることで，コンピュータは日常生活の基本的な枠組みの中へ浸透していくようになった．

とはいえ，マイクロプロセッサの誕生が電卓用LSIをきっかけとしたように，初期におけるマイクロプロセッサの役割は，既存の制御機器の置き換えが主目的だった．この場合，とにかく動くことが第一で，プログラムの生産性や性能は2の次だった．

小型で動けばいいという時代を経ると，当然のことながら，マイクロプロセッサは性能を要求されることになる．そこで，マイクロプロセッサは8，16，32とビット数を増やしながら大型計算機の進歩を大急ぎで追いかけていった．そして，現在の32ビットマイクロプロセッサの処理能力は大型計算機の処理能力に近づき（ある意味では凌駕し），コンピュータごとに専用のMPUを使用していたミニコンのMPUすら駆逐してしまった感がある．また，その応用分野もエンジニアリングワークステーション，画像処理システム，音声処理システム，ロボット制御，プロセス制御，人工知能システムなどという多種多様の分野に広がるようになった．

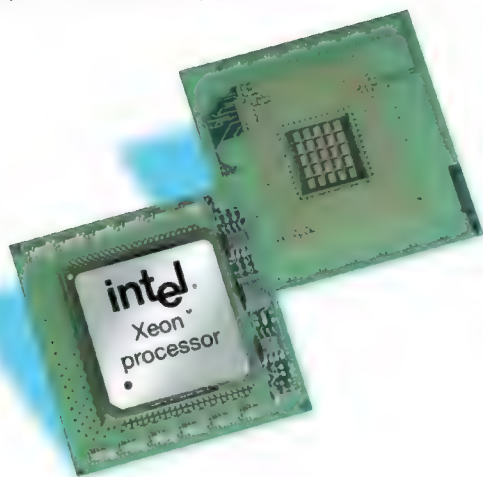
● もし……の世界，世界最初のMPUは日本製だったかもしれない

上述のとおり，MPUの歴史は，インテル社4004の1971年の発売に始まったといわれている．しかし，4004に遅れること数か月，日本でもμPD700という2チップ構成のMPUが開発されていることはあまり知られていない．これは，シャープがコカ・コーラ社から依頼を受けて論理設計をし，NECが製造した4ビットのプロセッサである．シャープは，最初，三菱電機に製造を依頼したのだが，これが見事にコケてしまう．そして，NECにお鉢が回ってきて日の目を見たわけだ．もし，シャープが初めからNECに依頼をしていれば，世界最初のMPUは日本製ということになっていたかもしれない．

μPD700は，その後NECが権利を買い取り，1チップのμCOM4として発売された．このMPUは電子式キャッシュレジスタを中心に広く応用されたという．筆者は命令セットの詳細をよく知らないが，どちらかといえばインテルよりもモトローラのMPUに近く，使い勝手が良かったと聞く．

4ビットMPUは，その後の低価格化の要求から，1チップマイコンが市場の中心になっていき，家電製品に採用されるようになる．この分野は日本の独壇上である．1チップマイコンは4ビット，8ビットと独自の進化を遂げていくが，16ビット以降になるとアメリカ製のメジャーなMPUに置き換えられていく運命をたどった．

日本の半導体史においては，国産第一号のマイコンは1973年に発表された東芝製のTLCS-12ということになっているもの

〔写真C〕Intel Xeon プロセッサ

もある．これは最初から12ビットプロセッサであったことが画期的である．インテルでは8ビットの8008が発表されたばかりである．さて，TLCS-12は米国フォード社の思惑が絡んで開発された．1970年に米国では自動車の排ガスを規制するマスキー法が成立し，自動車メーカーは対応を迫られていた．東芝とフォードは協力してエンジン制御の自動化をマイクロプロセッサに託したのだ．

● RISCの台頭

1980年頃，米スタンフォード大学とカリフォルニア大学のバークレー分校においてRISCの研究がなされていた．RISCとはReduced Instruction Set Computer（縮小命令セットコンピュータ）の略で，命令体系を単純化することで，それを実行するハードウェアも単純化し，高い動作周波数で高性能を得るという思想に則ったコンピュータのことである．スタンフォード大学やバークレー校の研究もその例から漏れていない．その共通するアーキテクチャは次のようなものだった．

- ロード/ストアアーキテクチャ
- パイプラインを用い，命令を1サイクルで実行
- 遅延分岐

これは，現在のRISCチップにみられる特徴でもある．RISCでは，インタロック（パイプラインのステージ間の待ち合わせ）などの複雑な制御をハードウェアで行わないことが基本である．それによって，ソフトウェアが複雑になってもハードウェアをできるだけ簡単にすることを第一としていた．これは，コンパイラ技術の劇的な進歩を呼んだ．まがりなりにも，RISCというMPUが使い物になることが世間に認められたのは，コンパイラ技術の向上によるところが大きい．

優れたコンパイラ技術に支えられたRISCの性能は，驚くべきものだった．たとえば，MIPSの最初のRISCであるR2000は1986年に発表されたが，それを採用したSGIのGWS（グラフィクスワークステーション）はわずか8MHzという低い動作周波数にもかかわらず，当時の32ビットMPUを採用したEWS（エンジニアリングワークステーション）以上の性能を発揮していた．RISCの高性能は徐々に世間に認められるようになり，現在ではほとんどすべてのMPUがRISCになっている．インテルやAMDが開発しているx86プロセッサも命令体系自体は従来どおりのCISCのものを採用しているが，その中身はRISC技術を最大限に採用して高速化を実現したものだ．

まとめ

原稿執筆段階ではもっと長かったのだが，本編がなかなか始まらないという声もあるので，プロローグはここまでとしよう．

人類は，その夢と理想をマイクロプロセッサという数ミリ角のチップに詰め込んできた．約30年前に初めて発表されたマイクロプロセッサは4ビットの処理能力しかなかったが，マイクロプロセッサは8ビット，16ビット，32ビット，64ビットと性能向上を達成してきた．いま，コンピュータの世界では32ビットが常識で，64ビットへの転換期にある．

ここ数年の動向を眺めていると，かつての大型計算機のMPUはEWSのMPUに駆逐され，そのEWSのMPUは（POWERやSPARCが気を吐いているものの）PCのMPUであったx86系MPUにとって変わられようとしている．これはPCもEWSも性能差がなくなってきていることを意味する．MPUの発展は，始まったばかりなのか絶頂期なのか……神ならぬ身の知る由もなし．

アカルサハ　ホロビノ姿デアラウカ
人モ家モ　暗イウチハマダ　滅亡セヌ

太宰治『右大臣実朝』より

参考文献

1) T・R・リード，『チップに組み込め！マイクロエレクトロニクス革命をもたらした男たち』，草思社，1986年
2) 嶋正利，『マイクロコンピュータの誕生　わが青春の4004』，岩波書店，1987年
3) 星野力，『だれがどうやってコンピュータを創ったのか？』，共立出版，1995年
4) 伊藤智義，久保田眞二，『BRAINS－コンピュータに賭けた男たち－(1)』，集英社，1996年
5) 嶋正利，「技術開発と教育」，『Interface』，2002年6月号

なかもり・あきら　フリーライタ

プロセッサの構成要素と動作の基本

プロセッサの基礎知識

中森　章

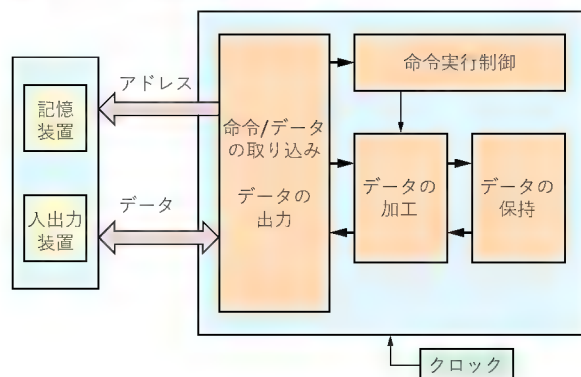
プロセッサって何だろう．専門書や教科書を読んでも難しそうだな．しかしMPUって，そんなに複雑なものだろうか．じつは，その背景にある考え方は，単純なのではないだろうか．直感で理解できたら嬉しいな．それが，今回の特集のテーマである．まずは，MPUの基本的な動作について解説する．（筆者）

1 MPUの構成要素

現代のMPUは**フォン・ノイマン型**と呼ばれる．これは**プログラム内蔵方式**のことであり，フォン・ノイマンが提唱したことになっているが，最近ではそれは誤りとされている．フォン・ノイマン型とかフォン・ノイマンボトルネックという言葉はやがて消滅するかもしれないが，ここでは慣例にしたがっておこう．

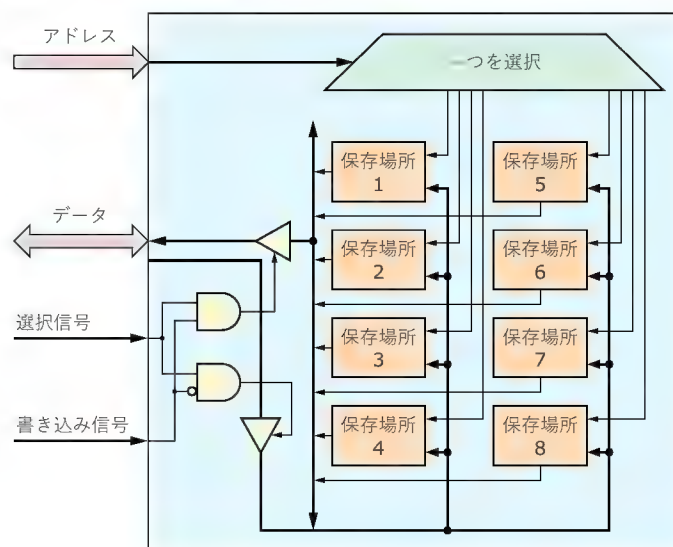
典型的なMPUは，「記憶装置」，「命令やデータを取り込むしくみ」，「命令実行を制御するしくみ」，「データを加工（処理）するしくみ」を基本的な構成要素とする．また周辺機器とデータをやりとりするための「入出力処理」というものもある．**図1**に典型的なMPUの構成要素を示す．

● プログラム内蔵方式には記憶装置が必須

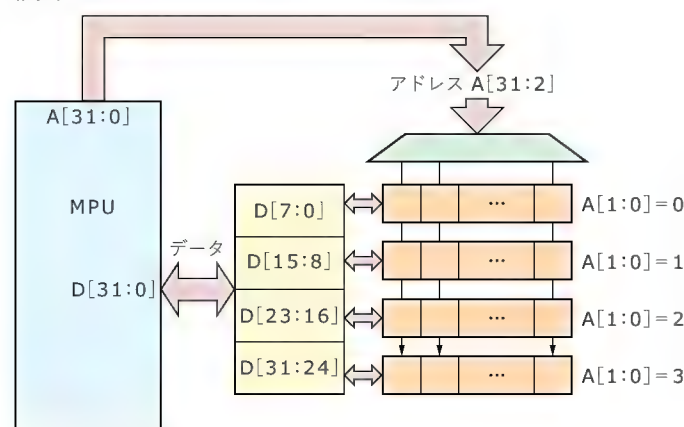
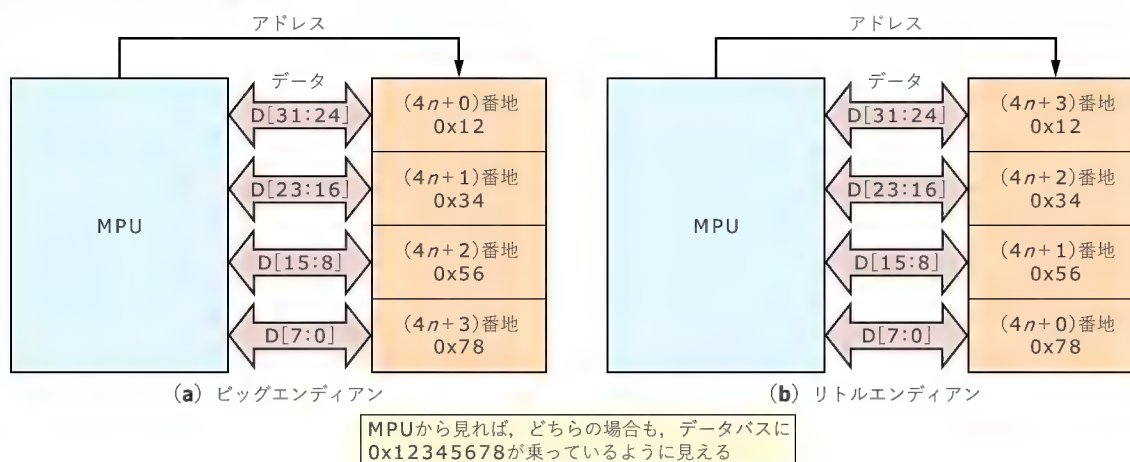
プログラム内蔵方式は，プログラムを内蔵するための**記憶装置**が必須である．ほかの構成要素はMPUに内蔵されるが，記憶装置は，基本的には，MPUの外部にある．この記憶装置は**メモリ**と呼ばれ，その構成により**ROM**（Read Only Memory）と**RAM**（Random Access Memory）に大別される．

メモリとは，複数の保存場所の集合で，各保存場所には位置を特定するための**アドレス**が付けられている．そして，あるアドレスを指定すると，それに対応する保存場所の内容が外部に読み出されるという装置である．RAMでは，アドレスと新しいデータを与えることで，アドレスで指定される保存場所の内容を変更することもできる．**図2**にメモリの概念図を示す．

メモリ内の保存場所の大きさ（ビット数）はいくつでもかまわない．しかし，最近のメモリは一つの保存場所の大きさを8ビット（＝1バイト）とする**バイトアドレス方式**が主流である．つまり，一つのアドレスを与えると1バイトのデータを得ることができる[注1]．

〔図1〕MPUの構成要素

〔図2〕メモリの概念図

注1：メモリによっては保存場所の大きさが16ビット（×16という），32ビット（×32という）のものも存在する．どちらかといえば，16ビットが一般的かもしれない．だたし，その場合もバイト単位での書き込みは可能になっている．

しかし，MPUの扱うデータは1バイトのみとは限らない．ハーフワード（16ビット），ワード（32ビット），ダブルワード（64ビット）といったデータ注2も扱う．おそらく，もっとも多用されるデータ長はワード（16ビットまたは32ビット）であろう．「○○ワード」という表現は，ワードが基準になっていることを示す証拠である．このため，MPUはある程度まとまったビット数（あるいはワード単位）のデータをメモリから取り出すほうが効率的である．このため，MPUとメモリ間のデータのインターフェース（データバス）は16ビット幅または32ビット幅であることが多い（アドレスのビット幅はまちまち）．たとえば，データバスが32ビット幅の場合は，1バイト出力のメモリを4個並列につなげて32ビットのデータ供給を実現できる（**図3**）．

アドレスはバイト位置を示すものである注3のに，一つのアドレスから4バイト（32ビット）のデータを取り出そうとすると「バイト並び（エンディアン）」の問題が生じる．複数バイトから構成されるデータに対し，バイトアドレスのより小さい保存場所にデータの上位の値を置くか，データの下位の値を置くかで2通りの方法がある．アドレスの小さい場所にデータの上位を置くのが**ビッグエンディアン**，逆にデータの下位を置くのが**リトルエンディアン**である．メモリでのバイト並びは異なるが，ビッグエンディアンでもリトルエンディアンでも，データバス上では同じイメージになる（**図4**）．

〔図3〕32ビットデータバス

MPU内の演算器などはデータの下位側から計算をしていくので，その意味で，すべてはリトルエンディアンに集約されるといえなくもない．エンディアンとは，あくまでもメモリ上にデータがどの順序で格納されているかを示しているにすぎず，MPUの内部処理とは直接は関係ない．また，ビッグエンディアンの場合，ビット番号の名付け方がリトルエンディアンと逆順になっていることが多く，惑わされやすいが，実質（バイト内のビットイメージ）は同じである．

● 命令やデータを取り込むしくみ

MPUがまず行わなければならないことは，メモリに格納された命令やデータを内部に取り込むことである．そのためには，メモリに与えるアドレスを生成し，メモリから出力されたデータを取り込めばよい．

まず，命令について考えよう．MPUは**PC**（Program Counter：プログラムカウンタ）という記憶機構（レジスタ）を備えている．これは，命令を取り込むメモリのアドレスを保持する．PCの値はアドレスバスを通じてMPUの外部に出力され，これがメモリに入力される．アドレスバスに値を出力すると，データバスを入力状態にしてメモリから出力された値（命令）を取り込

〔図4〕エンディアンとデータバスの関係

注2：インテルやモトローラに代表されるCISC時代からMPUを使ってきている人は，16ビットをワード，32ビットをダブルワード，64ビットをクオドワードと呼ぶ．現在は32ビットMPUが主流なので32ビットをワードと呼ぶのが自然だが，16ビットMPU時代の慣習も根強く残っている．CISCメーカーは16ビットをワード，RISCメーカーは32ビットをワードと呼ぶ傾向が強い．

注3：昔の大型計算機などではアドレスの割り付けがワード単位になっているものもある．つまり，すべてのデータはワード単位でしか扱わないのが基本である．このような場合にはエンディアンの問題はない．

む．これを**命令フェッチ**という．

MPUがリセットされると，ある特定の値がPCの初期値として設定される．そしてPCは，通常はメモリから取り込んだ命令のバイト数だけ値が増加していく．メモリから取り込んだ命令が分岐命令だった場合は，分岐先のアドレスが新たなPCとして設定される[注4]．

データバスから取り込まれた命令は，一般に，命令レジスタと呼ばれる記憶機構に保持される．それ以降の命令処理は，命令レジスタの内容にしたがって行われる．

さて，命令が扱うデータについて考えよう．メモリには命令のほかにデータも格納されている．命令によっては，その実行のためにメモリのデータが必要である．このため，命令の実行にともなって，メモリに格納された値が必要になった場合に活性化される機構を備える．メモリを参照するアドレスはロード命令やストア命令などを実行（正確にはデコード）することで生成される．このアドレスは**オペランドアドレスレジスタ**（とりあえず，そう命名する）という記憶機構に保持される．オペランドアドレスレジスタの値はアドレスバスを通じてMPUの外部に出力され，これがメモリに入力される．アドレスバスに値を出力すると，データバスを入力状態にしてメモリから出力された値（データ）を取り込む．これを**オペランドフェッチ**，または，**オペランドリード**という．

図5にMPUの命令やデータを取り込むしくみを示す．

● 命令実行を制御するしくみ

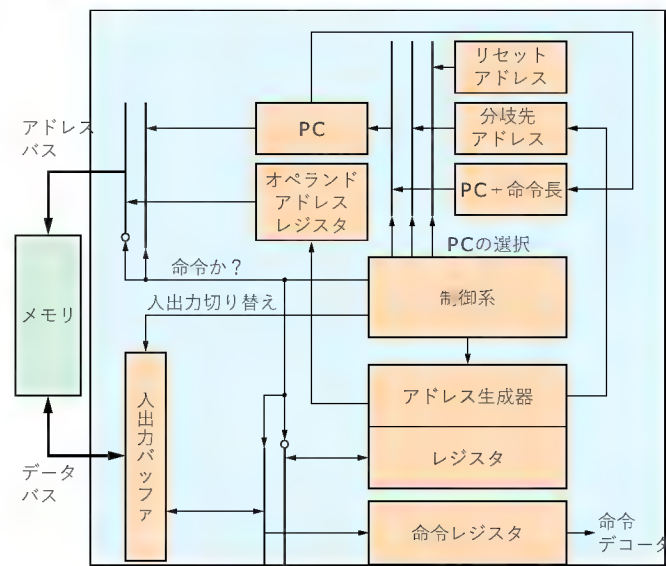
MPUは，メモリから取り込んだ命令を解釈し実行する．メモリから取り込まれた命令は**命令レジスタ**に保持され，それが**命令デコーダ**によって解釈（デコード）される．デコードとは命令コードに含まれているMPUに対する指令を取り出す機構である．具体的には，命令の種類を判別し，取り出した情報に基づいて，実行すべき演算の種類を決定したり，必要な制御信号を生成したりする．

命令がデコードされると，命令ごとにその後の処理手順が決定される．命令の実行とは，MPUの内部状態が変化することであり，ある状態になると回路が特定の状態（たとえば，入力を演算器に入れるとか）に変化する，またある状態では回路が別の状態（たとえば演算器の出力を取り出すとか）に変化する，ということを繰り返すことで実現される．この命令実行は，一般に，**クロック**と呼ばれる周期的に変化する信号に同期して行われる．クロックが進むにつれて，内部状態は，ある命令では，

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$

と状態変化をし，また，ある命令では，

$S_0 \rightarrow S_4 \rightarrow S_5 \rightarrow S_5$

と状態変化をする．ここでいう内部状態とは，具体的には，MPU内の各ゲートをON/OFFする組み合わせを示す．ある内部状態は，命令デコード結果と命令実行の中間結果を受けて，次にどの内部状態になるかが決定される．この状態変化によって，MPU内をデータが流れていく〔**図6**(**a**)〕．

〔**図5**〕命令やデータを取り込むしくみ

以上は命令デコード後の制御であるが，MPU全体も，

命令取り込み→命令デコード→命令実行

という大きな状態遷移をしながら制御されている〔**図6**(**b**)〕．

要するに，命令実行は状態遷移の塊なのである．このような状態遷移を司る機構を，**ステートマシン**，あるいは，**シーケンサ**と呼ぶ．つまり，MPUの実行とは，大小さまざまなシーケンサが組み合わされて複雑な状態遷移を行うことで実現されるのである．

● データを加工（処理）するしくみ

命令の実行とは，メモリに格納されているデータに対して何らかの加工（何もしないことを含む）をして，メモリに書き戻したり，命令実行の状態を変化させたりすることである．

メモリから取り込まれたデータは，レジスタと呼ばれるMPU内部のメモリに一時的に退避されることもある．MPUは，レジスタ（内部メモリ）やメモリ（外部メモリ）からのデータを適宜**演算器**に与えることでデータを加工する（**図7**）．

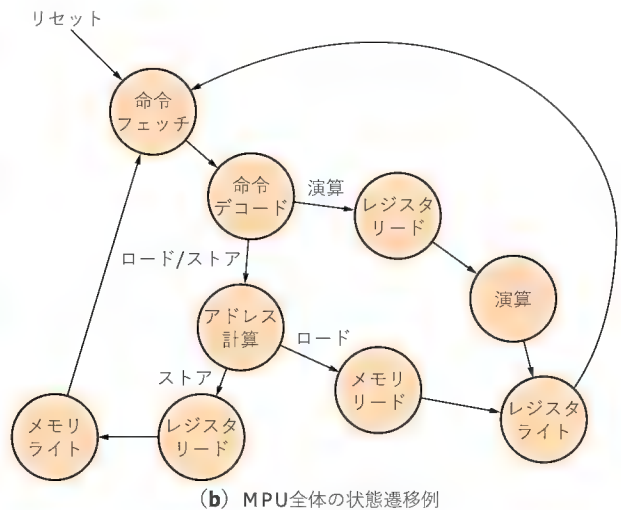
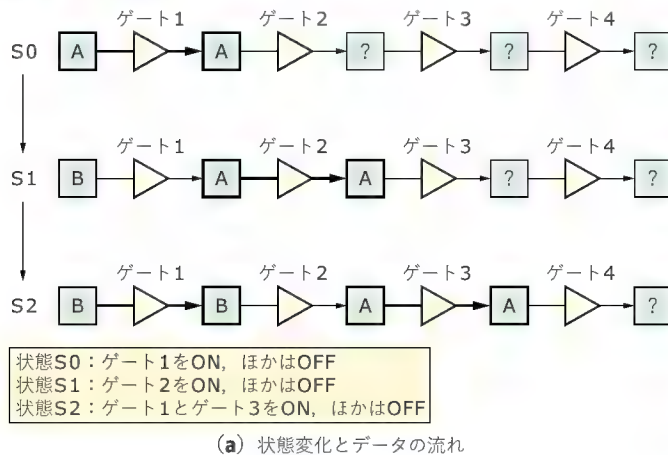
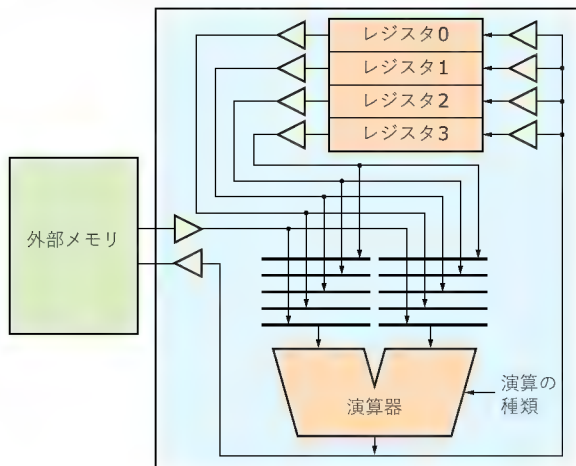
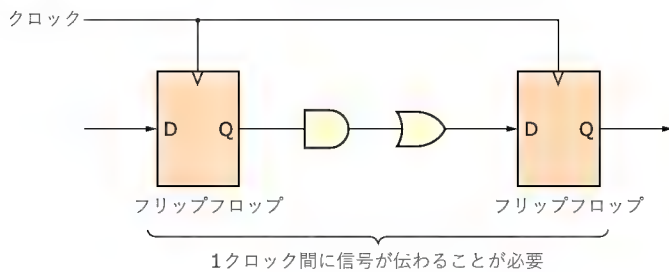
演算器のことを**ALU**（算術論理ユニット：Arithmetic Logic Unit）と呼ぶ．これは，加減算を行う算術ユニット（Arithmetic Unit）と論理和，論理積，排他的論理和などを行う論理ユニット（Logic Unit）の総称である．基本的な演算のうち，乗算と除算は専用のユニットで実現される．

どのメモリから演算器の入力をもってくるかという点は，MPUのアーキテクチャ（設計思想）に大きく関係する．基本的

注4：分岐先のアドレスは，通常，分岐命令の実行によって決定される．つまり，PCを分岐先に設定し直すタイミングは，分岐命令の実行後である．しかし，最近のMPUでは命令のフェッチと実行部分が切り離されている場合もあり，この場合は命令フェッチ部が自律的に分岐命令を処理する．

〔図6〕MPU内部の状態遷移

(a) 状態変化とデータの流れ

(b) MPU全体の状態遷移例

〔図7〕データを加工するしくみ

〔図8〕回路の模式図

には，二つ（同一でもいい）のレジスタから入力データをもってくる．しかし，

メモリ　と　レジスタ
レジスタ　と　メモリ
メモリ　と　メモリ

といった入力の組み合わせも当然考えることができる．ソフトウェアの作りやすさを考慮すれば，すべての組み合わせが可能なほうがプログラミングの自由度が高く嬉しい．ハードウェアの設計しやすさを考慮すれば，メモリから取り込んだデータは必ずレジスタに格納することとし（これは何もしないという処理），演算器の入力はレジスタのみに限定したほうがハードウェア設計が楽で嬉しい．

前者は**CISC**の考え方であり，後者は**RISC**の考え方である．ただ，二つの入力がどちらもメモリというのは，制御がかなり複雑になるため，CISCであっても，入力の少なくとも一方はレジスタに限定されていることが多い．

演算器での演算の種類は，命令デコードによって決定される．なお，**図5**にあるアドレス生成器も演算器の一種であり，その実体は加算器である．アドレス生成器は，専用にもつ場合と，通常の演算器で代用する場合がある．

● クロック

MPUの動作を理解するとき，クロックの存在を忘れることはできない．クロックとは一定の周期で0と1を繰り返しながら自走している特殊な信号である．これは，MPUの動作タイミングの基準となる．1秒間に1回だけクロックが0から1に変化する（1周期）速さを1Hz（ヘルツ）という．最近のMPUは800MHzとか1GHzというクロックで動作するが，800MHzとは1秒間に800×M（メガ：100万）＝8億回，1GHzとは1秒間に1×G（ギガ：10億）＝10億回クロックが変化することを意味する．

MPUの内部回路はフリップフロップの集合で構成される．フリップフロップとは，クロックの切り替わり時に新たな状態（0または1という信号）を保持する回路である．フリップフロップを複数個並列に並べたものがレジスタである．

MPUの回路を，フリップフロップとフリップフロップ間を配線したものとして模式化すると（**図8**），回路が動作するということは，クロックが1回変化する間に前段のフリップフロッ

プに保持した値が，銅線を通じて，後段のフリップフロップに保持し直されることに相当する．レジスタなどを考える場合，フリップフロップ間の配線長は完全には同じでないので，後段のフリップフロップに信号が到達する時間は同一ではない．あるいは，フリップフロップ間には論理積や論理和のゲートが挿入されていて，ゲートを通過するごとに少しずつ信号が遅延する．クロックとは，その信号の到達時間を規定するものである．つまり，1クロック間に信号が伝わらないと誤動作する．全部の信号が正しく伝わるためのクロックの最高周期が，MPUの最高動作周波数である．

● 汎用レジスタ

MPU内部のメモリをレジスタということはすでに説明した[注5]．レジスタは演算器の入出力となるデータの一時記憶場所である．レジスタは演算器のデータとなるほかに，アドレッシングモードにおいて，レジスタ間接用のベースアドレスやインデックスを与えるためにも用いられる．このようにレジスタの用途はいろいろあるが，すべての用途に使用できるレジスタを**汎用レジスタ**(General Purpose Register)という．

最近のMPUの提供するレジスタは汎用である．このため，最近のMPUは，**汎用レジスタ方式**と呼ばれる．しかし，昔はハードウェアを簡略化するために，目的別の専用レジスタを用意するものもあった．つまり，ベースアドレス用，インデックス用というようにレジスタの用途が限られていた．このような場合，演算器の入出力となるレジスタも限られている．具体的には，演算器の入力の一つと出力が演算用レジスタ(アキュムレータ)に接続されている．演算器のもう片方の入力が汎用で，ほかの専用レジスタやメモリ，アキュムレータ自身に接続されている．このような方式は**アキュムレータ方式**と呼ばれる．

● 入出力

入出力とは，たとえばx86系のMPUではMOVという転送命令のほかにINやOUTという入出力のための命令が存在する．MOVはメモリやレジスタに対するデータの入出力を司るので，新たに入出力といわれてもピンとこない．ここでいう「入力」と「出力」とは周辺装置からの情報の入力，周辺装置への情報の出力を示す．しかし，これらの命令が実際に行う処理は，あるアドレスからデータを入力すること，あるいは，あるアドレスへデータを出力することである．それではますます，MOVとの違いがわからない．じつは「入出力」命令でのアドレスはメモリではなく，周辺装置に直接接続されているのである．メモリと区別する意味で，「入出力」命令でアドレスする(指し示す)対象を**I/Oポート**という．

通常のMOV命令とIN/OUT命令を区別する場合，アドレスとデータの入出力時には，それがメモリかI/Oポートであるかを示す信号(外部端子)が使われる．MPUを使ったシステムを構築する場合は，このメモリなのかI/Oなのかの信号を見て参照する対象を振り分けられるようにしなければならない．

MPUによっては専用の「入出力」命令を提供していないものもある．これはメモリ空間の特定位置をI/Oポートとみなす方式である．これを**メモリマップトI/O**という．

I/Oポートとメモリの差異は，その逐次性にある．メモリに対しては投機実行やアウトオブオーダ実行が考えられるが，I/Oにはそれがない．これらについては，後述する各章で解説する．

2 命令コード，オペランド，アドレッシングモード

● 命令の形式

ここではMPUが処理する命令に関して考える．上述のように，命令はデコードされることによって処理に関する情報を抽出する．逆にいえば，命令には処理に関する情報が符号化(エンコード)されている．命令は数値の形態でメモリに格納されている．この意味で命令とデータに区別はない．そして，命令を示す数値をビットで表すと，それぞれのビットが意味をもっている(情報がエンコードされている)ことがわかる．命令を示す数値を**命令コード**と呼ぶ．

一つの命令コードのビット数は何ビットでもかまわないのだが，メモリに効率良く格納できるように1バイト(8ビット)の倍数が用いられる．CISCでは命令の種類によって命令コードのビット長がバイト単位で可変になっていたりするが，RISCでは命令デコードを簡単に行うために固定長(16ビット，あるいは，32ビット)であることが多い．

一般に命令コードは，**オペレーションコード**(オペコード)と**オペランド**の二つの領域に分けられる．オペコードは命令の種類を示し，オペランドは扱うデータの形態を示す．オペランドは**アドレッシングモード**で規定される．アドレッシングモードとは，データがどこに格納されているかを示す形式である．アドレッシングモードの例としては，

- 即値(イミディエート)
  命令コードに埋め込まれた定数値
- レジスタ
  レジスタにデータがある
- 直接アドレス
  アドレスで示すメモリにデータがある
- レジスタ間接
  レジスタにあるアドレス値であるメモリにデータがある

注5：厳密にはレジスタとメモリは異なる．メモリにはフリップフロップで値を保持するSRAMとコンデンサの容量で値を保持するDRAMがあり，通常，MPU内部に搭載されるのはSRAM(キャッシュメモリなどに使用される)である．その意味で，どちらもフリップフロップの集まりであるが，メモリは専用に設計され，小面積で大容量の情報を記憶できる．つまり，レジスタをSRAMで構成すれば面積が縮小できるのだが，SRAMはクロックに同期して動くわけではないので，アクセス時間を規定しにくく回路設計が難しくなる．

- インデクスつきレジスタ間接
  レジスタにあるアドレス値にインデクスレジスタの値を加えたアドレスにデータがある
- ディスプレースメントつきレジスタ間接
  レジスタにあるアドレス値にディスプレースメントを加えたアドレスにデータがある

などがある．メモリ参照は，基本的には，

ベースレジスタ＋インデクスレジスタ
＋ディスプレースメント（オフセット）

ですべてが表せる．このとき，インデクスレジスタの値は参照するデータサイズにしたがってスケーリングされる（バイトなら×1，16ビットなら×2，32ビットなら×4，64ビットなら×8）こともある．つまり，インデクスの値は何番目のデータであるかを示す．これを**自動スケーリング**という．

また，**メモリ間接アドレッシング**というものもあり，これは，上述のアドレス計算で求められたメモリの内容を新たなベースアドレスとし，ディスプレースメントを加えてメモリアドレスとするものである．

なお，ベースレジスタとしてプログラムカウンタ（PC）を指定できる場合もある．これを**PC相対アドレッシング**と呼び，ポジションインデペンデントなオブジェクトコードを作成する場合に使用される．**図9**にアドレッシングモードの例を示す．

このように，アドレッシングに関してはいろいろな方法が考えられるが，CISCは豊富なアドレッシングモードを特徴とし，RISCは単純なアドレッシングモードを特徴とする．具体的には，CISCは何でもありで，RISCは即値，レジスタ，ディスプレースメントつきレジスタ間接が典型的なアドレッシングモードである．

ところで，命令の実行は，二つのソースオペランドを入力とする演算を行い，結果をデスティネーションオペランドに格納する．二つのソースオペランドと一つのデスティネーションオペランドを独立に指定することができるのが**3オペランド方式**であり，一つのソースオペランドとデスティネーションオペランドが共通なものが**2オペランド方式**である．3オペランド方式では命令の中に三つのオペランド領域が存在し，2オペランド方式では命令の中に二つのオペランド領域が存在する．

オペコードのビット数は命令の個数を示す．つまり，2ビットなら4種類，3ビットならは8種類，4ビットならば16種類，5ビットならば32種類……，という具合である．命令コードのオペコード以外のビットはオペランドを示す．このビットが，基本的には，2オペランド方式では二つに，3オペランド方式では三つに分割される．オペランドのうち，アドレッシングモードに含まれるレジスタはレジスタの番号で示される．このレジスタ番号を示す領域のビット数は，MPUが備えるレジスタの本数によって決定される．4本なら2ビット，8本なら3ビット，16本なら4ビット，32本なら5ビット，……，という具合にビットが必要である．また，オペランド領域の分割のやり方は，命令の種類やアドレッシングモードによって少しずつ異なる．この違いを**命令形式**（フォーマット）という．命令形式の例を**図10**に示す．

以上の説明からわかるように，命令コードのビット長は，オペコードの種類，アドレッシングモードの種類，レジスタの本数，オペランドの数などで決定される．これらを総称して**命令セットアーキテクチャ**という．MPUでは，無限に長い命令コードを使

〔図9〕メモリアドレッシングの例

〔図10〕命令形式の例

(b) 形式1 ディスプレースメントつきレジスタ間接ロード/ストア命令

4	3	3	6
オペコード	レジスタ	レジスタ	ディスプレースメント

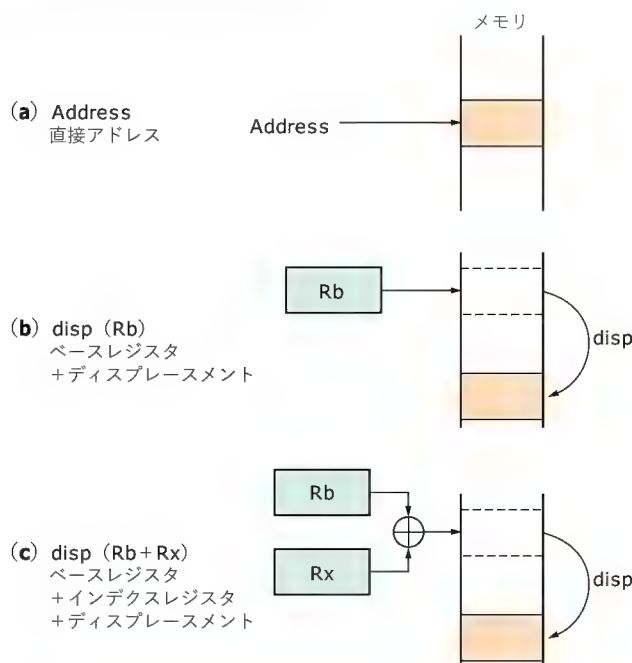
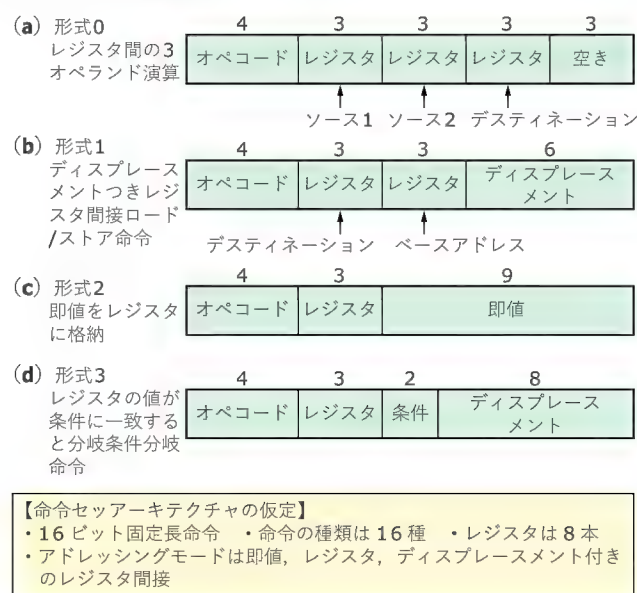
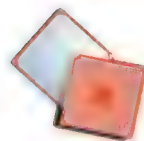
デスティネーション　ベースアドレス

(c) 形式2 即値をレジスタに格納

4	3	9
オペコード	レジスタ	即値

(d) 形式3 レジスタの値が条件に一致すると分岐条件分岐命令

4	3	2	8
オペコード	レジスタ	条件	ディスプレースメント

【命令セッアーキテクチャの仮定】
- 16ビット固定長命令　・命令の種類は16種　・レジスタは8本
- アドレッシングモードは即値，レジスタ，ディスプレースメント付きのレジスタ間接

用できるわけではないので，その命令セットアーキテクチャで命令コード長が決定される．逆にいえば，固定長の命令コードを採用する場合は，命令セットアーキテクチャを詳細に検討しなければ，情報が命令コードに入りきらなくなってしまう．

● 命令の流れと実行

MPUの構成要素がわかったところで，それらがどのような係わりをもって動作するのかを見ていこう．**図11**(pp.48-49)に模式化したMPUのブロック図と，演算命令/ロード命令/ストア命令に関し，各状態での命令とデータの流れを太線で示す．もちろん，MPUにはこれら以外の命令も存在するが，ここでは省略する．命令フェッチと命令デコードはすべての命令で共通であるが，それ以降は命令デコード結果によって異なる状態遷移をする．

(1) 演算命令(レジスタ-レジスタ演算)

演算命令は，命令フェッチ〔**図11(a)**〕，命令デコード＆レジスタリード〔**図11(b)**〕，命令実行〔**図11(d)**〕という状態遷移を行うことで命令を実行する．

(2) ロード命令

ロード命令は，命令フェッチ〔**図11(a)**〕，命令デコード〔**図11(c)**〕，アドレス計算〔**図11(e)**〕，メモリアクセス1＝オペランドリード〔**図11(f)**〕，メモリアクセス2＝レジスタライト〔**図11(h)**〕という状態遷移を行うことで命令を実行する．

(3) ストア命令

ロード命令は，命令フェッチ〔**図11(a)**〕，命令デコード〔**図11(c)**〕，アドレス計算〔**図11(e)**〕，メモリアクセス1＝レジスタリード〔**図11(g)**〕，メモリアクセス2＝オペランドライト〔**図11(i)**〕という状態遷移を行うことで命令を実行する．

● パイプライン

昔のMPUは一つの命令の実行が終わった後で，次の命令の実行を開始していた．しかし，命令実行に係わる状態遷移において，演算器など多くのハードウェア資源は1回しか使用されない．これでは，あまりにも効率が悪い．命令実行の状態をオーバラップさせることで，クロックごとにハードウェア資源を使用することができ，命令実行のスループット(クロックごとに実行が終了する命令の個数)も向上する．これがパイプラインの考え方である．

パイプライン構造でMPUを動作させるためには，各状態で演算結果などのデータを保持しておけばよい．**図12**(p.49)に，演算命令をパイプライン構造で実行する場合の，ハードウェア資源の使用状況を示す．

パイプラインに関しては，第2章および第3章で詳細に解説する．

● キャッシュ

これまでの説明では，メモリを1クロックで参照できるものとして話をすすめてきた．しかし，現実的には，メモリのアクセス時間はMPUのクロックと比べて非常に遅い．実際の命令実行では，メモリを参照する状態(命令フェッチ，オペランドリード，オペランドライト)は複数のクロック数を消費する．これでは，命令実行がメモリのアクセス時間に律速されてしまい，高速な実行ができない．

そこで考案されたのが，従来は外部にあったメモリをMPUの内部に取り込むことである．この場合，高速なSRAMを集積することが可能になり，1～2クロックでメモリを参照できる．これが**キャッシュメモリ**である．しかし，外部メモリと同じ容量のキャッシュメモリを内蔵することは不可能である．そこで，外部メモリの一部分をコピーしてキャッシュメモリに格納し，適宜外部メモリとの入れ替えを行って，高速な命令実行を維持する．

キャッシュメモリに関しては，次号で詳細に解説する予定である．

● MMU(仮想記憶)

命令やデータを格納する外部メモリの容量には限界がある．しかし，ソフトウェアの高度化にともない，プログラムがメモリに格納しきれない場合も出てきた．これを解決するために，プログラムを分割してハードディスクなどの補助記憶装置に格納し，少しずつメモリの内容と置き換えながら，それを実行するという方式が考えられた．発想はキャッシュメモリと同じである．このためには，プログラムで参照するアドレス(仮想アドレス)を実際のメモリのアドレス(物理アドレス)に変換する必要がある．この機能を提供するのが**MMU**(Memory Management Unit：メモリ管理ユニット)である．

MMUには，マルチタスクの実現，メモリ保護の実現という機能も有している．MMUに関しては，次号で詳細に解説する．

● 割り込み

MPUの処理内容が高度化してくるとソフトウェアも複雑になる．そこで，プログラムの本筋とは直接関係のない処理を独立に行うという発想が生まれた．それを実現するのが割り込みである．割り込みが発生すると，MPUはそれまで実行していた処理をいったん中断して，割り込み処理という別の処理の実行を始める．割り込み処理が終了すると，プログラムの実行は割り込みが発生した時点から再開する．プログラムは本筋のプログラムと割り込み処理用のプログラムを独立に開発できる．割り込みを使用しない場合は，割り込み処理で行うような別処理を本筋のプログラムからサブルーチンコールをして実行させなければならなく，プログラム開発において，そのための余計な考慮が強いられる．

割り込みに関しては，次号で詳細に解説する．

● プログラムとは

MPUすなわちコンピュータは，メモリに格納された命令を取り込み，その指示するとおりに動作する．昔から，コンピュータはソフトウェアがなければ只の箱(粗大ゴミとも)といわれる．ソフトウェアとはプログラムのことであるが，それではプログラムとは何だろう．これは，今回の目的ではないので簡単に説明する．

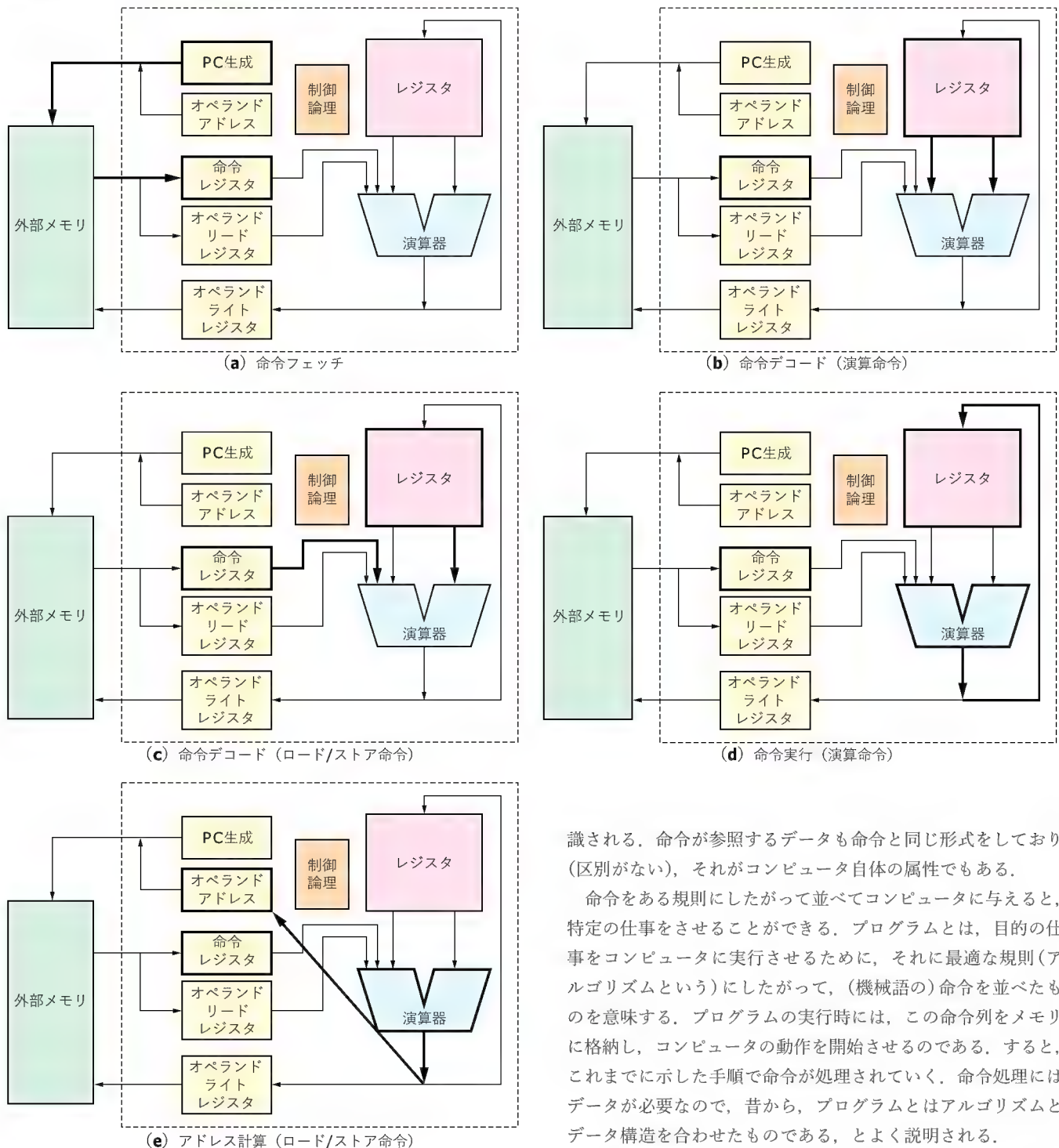
〔図11〕MPUのブロック図と命令実行の流れ

(a) 命令フェッチ

(b) 命令デコード（演算命令）

(c) 命令デコード（ロード/ストア命令）

(d) 命令実行（演算命令）

(e) アドレス計算（ロード/ストア命令）

コンピュータが理解できる命令は，コンピュータごとに定義された**機械語**と呼ばれるビット列である．一般的には，ビット列が8ビット（バイト），16ビット（ハーフワード），32ビット（ワード）の塊になってメモリに格納されており（そのほうがメモリにとって都合がよいので），それが機械語の命令として認識される．命令が参照するデータも命令と同じ形式をしており（区別がない），それがコンピュータ自体の属性でもある．

命令をある規則にしたがって並べてコンピュータに与えると，特定の仕事をさせることができる．プログラムとは，目的の仕事をコンピュータに実行させるために，それに最適な規則（アルゴリズムという）にしたがって，（機械語の）命令を並べたものを意味する．プログラムの実行時には，この命令列をメモリに格納し，コンピュータの動作を開始させるのである．すると，これまでに示した手順で命令が処理されていく．命令処理にはデータが必要なので，昔から，プログラムとはアルゴリズムとデータ構造を合わせたものである，とよく説明される．

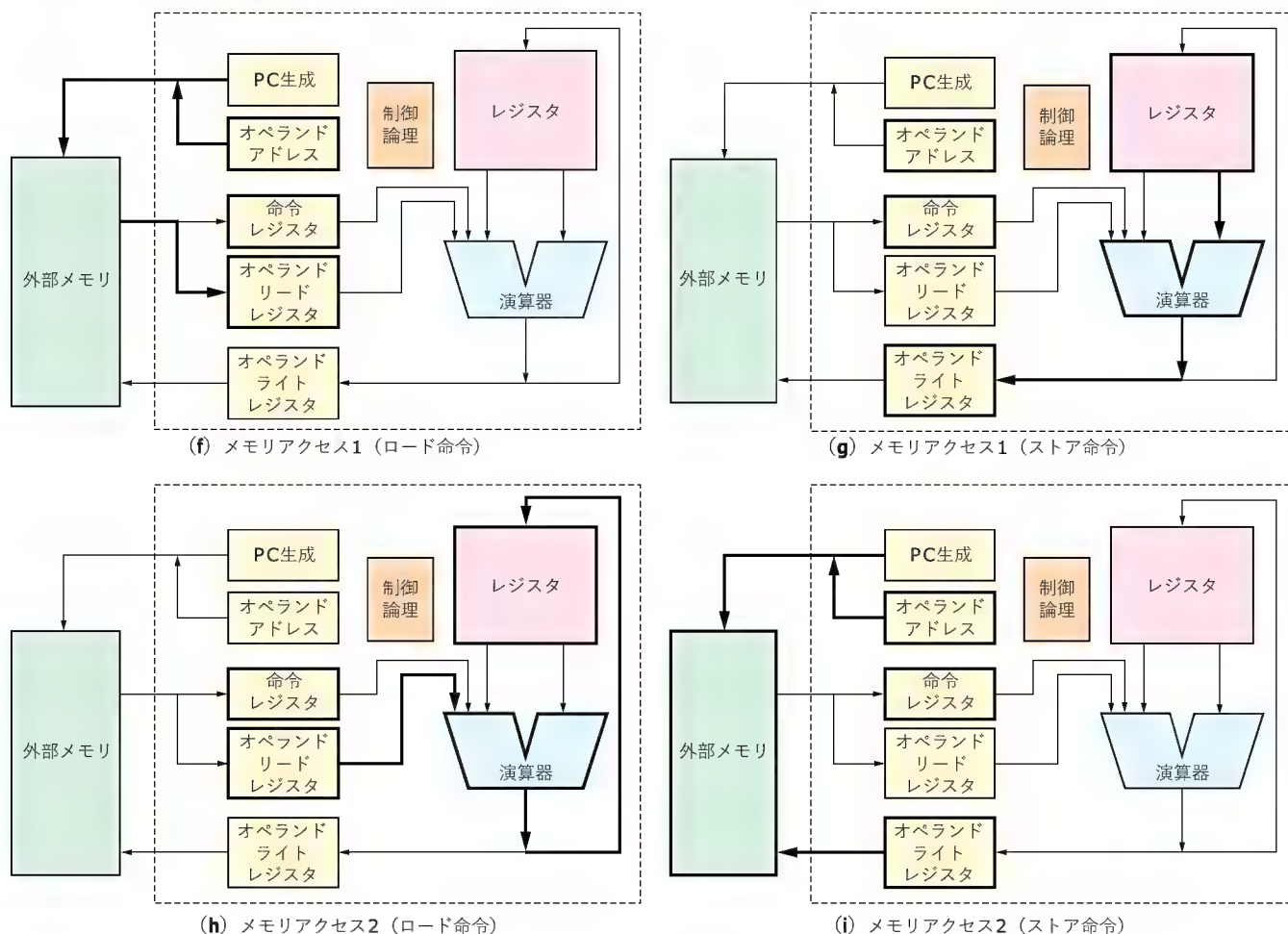
さて，機械語は0と1のビット列なので人間には理解しにくい．それを人間が理解しやすくするために意味をもった（英語に近い）記号に対応させて扱う．つまり，MOV（転送）とかADD（加算）といった記号で機械語を代表させる．これらの記号を**ニーモニック**と呼ぶ．ニーモニックを使用してプログラムを書くための手段が**アセンブリ言語**である．アセンブリ言語を機械語に変換するしくみ（これも結局はプログラムである）が**ア**

〔図11〕MPUのブロック図と命令実行の流れ(つづき)

(f) メモリアクセス1(ロード命令)

(g) メモリアクセス1(ストア命令)

(h) メモリアクセス2(ロード命令)

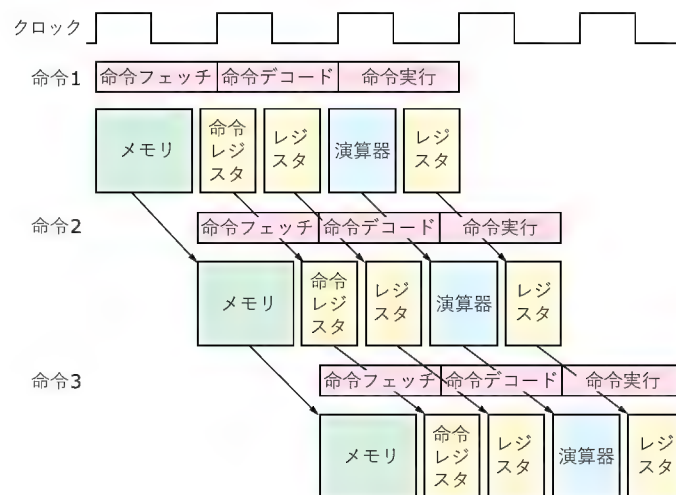
(i) メモリアクセス2(ストア命令)

センブラである.

アセンブリ言語は，人間に理解しやすくなっているとはいえ，所詮は機械語とほぼ1対1に対応しているので，機械語そのものである．個々の命令機能が単純すぎて複雑なアルゴリズムを記述するためには向いていない．そこで考案されたのが，人間の思考を反映させやすくした**高級言語**である．「高級」というのは「飛び抜けた」という意味ではない．言語仕様が機械語にどの程度近いかの指標であり，アセンブリ言語は低級言語といわれる．その対極としての「高級」である．

まとめ

この章を執筆していて，何だかコンピュータの教科書を作っているような幻想に見舞われた．コンピュータの教科書にはいろいろな知識が詰まっているが，その項目があまりにも多いので，ややもすると本質を見失いがちである．今回の目的はMPUのしくみを直感的に理解しようというものなので，2進数やブール代数といった一般の教科書に載っているような事項は解説しない．

〔図12〕パイプラインとハードウェア資源の供給

クロック
命令1　命令フェッチ　命令デコード　命令実行
メモリ　命令レジスタ　レジスタ　演算器　レジスタ
命令2　命令フェッチ　命令デコード　命令実行
メモリ　命令レジスタ　レジスタ　演算器　レジスタ
命令3　命令フェッチ　命令デコード　命令実行
メモリ　命令レジスタ　レジスタ　演算器　レジスタ

本章は，いわば導入あるいは概説である．次章から各論に入る．

なかもり・あきら　フリーライタ

Chapter 2

もっとも基本的なプロセッサ高速化技法

パイプライン処理の概念

中森　章

本章ではパイプラインについて説明する．パイプラインとはMPUの命令実行を高速化する手法の一つであり，現在では，ほとんどすべてのMPUで採用されている．本章では，一度に1命令を実行する通常のパイプライン処理について解説する．このパイプラインは，2命令以上を同時実行するスーパースカラに対して，とくにシングルパイプラインと呼ばれている．またはユニスカラパイプライン，あるいは単にスカラパイプラインと呼ばれることもあるようである．　　　　（筆者）

1 パイプラインとは

● 流れ作業＝パイプライン

コンピュータの性能を向上させる方法については，いろいろ考案されている．**パイプライン**とは，ハードウェアを並列化して性能を向上させるための一般的な手法である．その基本的な考え方は，プログラム内蔵方式を提唱したフォン・ノイマンによってすでに提案されていたという．たとえば，MPUの命令実行に比べて10倍以上も遅いメモリアクセスが存在する状況下で効率的に命令の処理を行うために，命令の実行とメモリアクセスをオーバラップして処理することが考えられた．これが，パイプライン処理の原型である．

パイプラインの基本的な考え方はごく自然なものである．なにもコンピュータの技術に固有なものではない．自動車の製造ラインや電子部品工場などで行われている流れ作業は，パイプラインそのものである．一つの製品が数分後ごとに完成していくようすを思い浮かべてほしい．実際，パイプラインの呼び名は，石油が次々とパイプを通過していく石油化学パイプラインと動作が似ていることに由来している．

各工程が1単位時間かかるN工程からなる処理を考える．単純に考えると，この処理を終了するためにはN時間を要する〔**図1(a)**〕．これをN人の人が流れ作業によって各工程を分担し，前の工程から受け継いだ製品に1単位時間で加工を施して，後の工程に引き継ぐようにする〔**図1(b)**〕．この場合，もともとの処理ではN時間に一つしか製品が完成しないが，流れ作業では見かけ上，1単位時間に一つの製品が完成することになる．つまり，処理速度はN倍に改善される．これがパイプラインの原理である．

● ステージ，段数，ハザード

ここで，各工程をパイプラインの**ステージ**という．「段」という表現も使われ，N工程から構成されるパイプラインはNステージパイプライン，またはN段パイプラインと呼ばれる．また，あるステージを分担する人が手間取って，そこでの処理を1単位時間以内に終わらせることができないような場合は，パイプライン処理に乱れが生じ，処理性能が低下する．パイプラインステージでの処理を単位時間内に終わらせることを阻害する要因を**ハザード**という．

〔図1〕パイプライン処理の概念

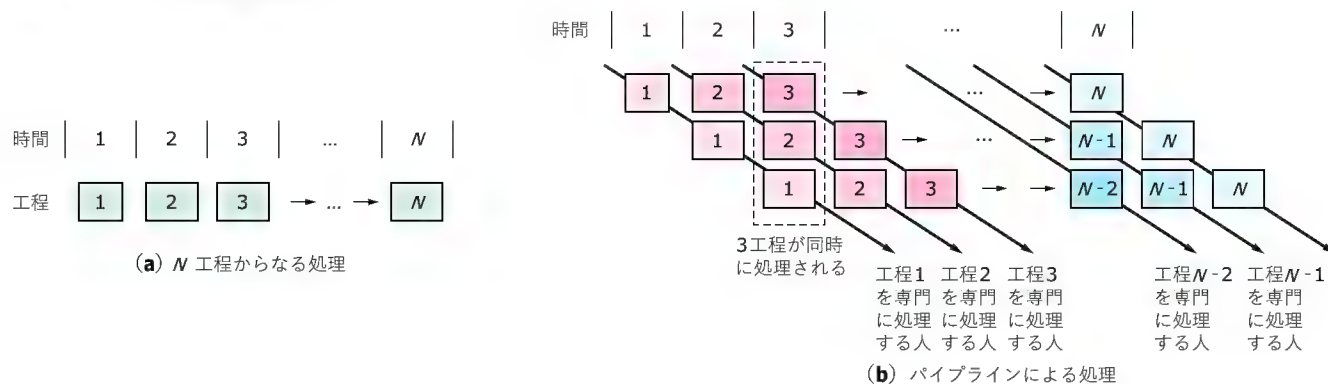
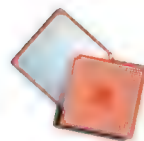
時間　1　2　3　…　N
工程　1 → 2 → 3 → … → N

(a) N工程からなる処理

時間　1　2　3　…　N

3工程が同時に処理される

工程1を専門に処理する人　工程2を専門に処理する人　工程3を専門に処理する人　工程N-2を専門に処理する人　工程N-1を専門に処理する人

(b) パイプラインによる処理

パイプライン処理をコンピュータに適用する場合は，各ステージが並列に処理できることが前提である．ハードウェア資源を共有するステージがあると，ハザードが生じ，待ち合わせが必要になる(これを**ストール**という)．逆にいうと，ハードウェア資源が競合しないようにパイプラインステージを分割することがプロセッサ設計者の腕の見せどころである．

パイプライン処理はまず，大型計算機で採用された．その後，半導体の集積技術が進み，MPUでも大量のトランジスタが利用可能になると，MPUにも採用されるようになる．パイプライン処理の採用を大々的に表明したMPUは，NECのV60が最初ではないかと筆者は記憶している．それ以前のインテルの8086でもオペランドフェッチと実行をパイプライン化していたが，インテルがパイプラインを明言したのは80386以後(最近のインテルの発言ではPentium以降)となっている．一方，68000系のMPUも古くからバスサイクル同期のパイプライン処理をしていたようである．しかし，こちらもパイプラインを明言したのは68060が初めてだったと思う．68060はすでにスーパースカラ構造になっていたので，シングルパイプライン時代の68000系のパイプライン構造は不明である．

〔図2〕RISCのパイプライン処理

(a) ステージと機能ブロックの関係

(b) スムーズなパイプラインの流れ

2 パイプラインの理論

● パイプラインステージ

CISC初期においてもパイプライン構造を採用しているものがあった．しかし，それらのMPUにおいてパイプライン処理は有効に機能していたとはいえない．各MPUメーカーがパイプラインを強調しなかったのは，それが性能に寄与していなかったからではないかと考えられる．しかし，RISCの登場によってパイプライン処理はにわかに脚光を浴びる．RISCのパイプラインは，CISCとは異なり，全命令でパイプラインのステージ数は固定であることが多い．筆者だけの感覚かもしれないが，命令フェッチ，命令デコード，実行という処理の流れも，その区切りが明確になっているように感じる．

RISCの存在意義は，パイプライン処理をいかに効率的に実現できるかにかかっているといっても過言ではない．このため，RISCでは命令やオペランドをキャッシュからフェッチすることを前提としている．通常のメモリはアクセス時間が遅いので，メモリアクセスステージの処理時間が他のステージに比べて長くなり，効率的なパイプライン処理を行うことはできない．ステージの処理時間を均一化するため，キャッシュの導入は必然だったといえる．キャッシュの導入により，メモリアクセスステージが1または2クロックという固定クロック数で処理できるようになる．

RISCのパイプラインは，コンピュータアーキテクチャの有名な教科書で学ぶことができる．それが，HennessyとPattersonによる『コンピュータアーキテクチャ』(通称ヘネパタ)である．この教科書では，仮想的なMPUとしてDLX(デラックスと発音する)というMPUを定義し，そのパイプラインとして次の5ステージの処理が提案されている．もっともDLXはMIPSのR2000/R3000と非常に近い(同じ？)構造をしており，以下はR3000のパイプラインそのものということもできる．ただし，ヘネパタではメモリがキャッシュであることをとくに強調してはいない．

● RISCのパイプライン処理

RISCのパイプライン処理を**図2**に示す．パイプラインがスムーズに動作する場合は，全ステージ数と同じ数の命令が(理論的には)同時実行できる．

(1) 命令フェッチ(IF)

命令キャッシュから命令を取り出す．

(2) 命令デコード(RF)

フェッチした命令をデコードする．同時にレジスタオペランドをフェッチする．

(3) 命令実行(EX)

デコード結果とフェッチしたレジスタの値を基に命令を実行する．ロード/ストア命令の場合は実効アドレスの計算を行う．分岐命令の場合は分岐先アドレスを計算する．

(4) オペランドフェッチ(MEM)

EX ステージで計算したアドレスに対応するメモリの値をデータキャッシュよりリードする.

(5) ライトバック(WB)

EX ステージで計算した結果, または MEM ステージでフェッチしたオペランドをレジスタに格納する. ストア命令の場合はデータキャッシュにライトする.

上のパイプラインではアドレス変換のステージがないが, これは IF または MEM ステージに先立って行われる. この詳しい説明は次章で解説する R3000 のパイプラインの実際の項に譲る.

RISC のパイプラインの特徴は, アドレス計算をする専用のステージがなく, EX ステージで代用している点である. このため, アドレス計算用の演算器と命令実行用の演算器(実際は加算器)をそれぞれ別個に用意する必要はない. これは RISC の「ロード/ストアアーキテクチャ」という特徴に由来する. つまり, 一つの命令では 2 回加算を行うことがない, 1 命令で 1 回だけ演算器を使用するという制限の下で, レジスタとレジスタ間の加算, または, アドレス計算(ロード/ストア命令)は別の命令に分かれて定義されている.

● データハザードとフォワーディング

パイプラインの処理が乱れるハザードは, RISC のパイプラインでも発生する. それを詳しく見ていこう. まずはレジスタの依存関係に起因するハザードである. レジスタ間のリード/ライトの前後関係で, 次の 4 種類が考えられる.

(1) RAW(Read After Write)ハザード

これは, レジスタライトの完了前に後続命令によって同一のレジスタをリードしようとした場合に生じる(**図 3**).

(2) WAR(Write After Read)ハザード

これは, レジスタから値をリードする前に後続命令によって同一のレジスタにライトをしようとした場合に生じる.

(3) WAW(Write After Write)ハザード

これは, 同一レジスタへのライト順序が狂う場合に生じる.

(4) RAR(Read After Read)ハザード

一応挙げたが, レジスタへの変更がともなわないので, このようなハザードは存在しない.

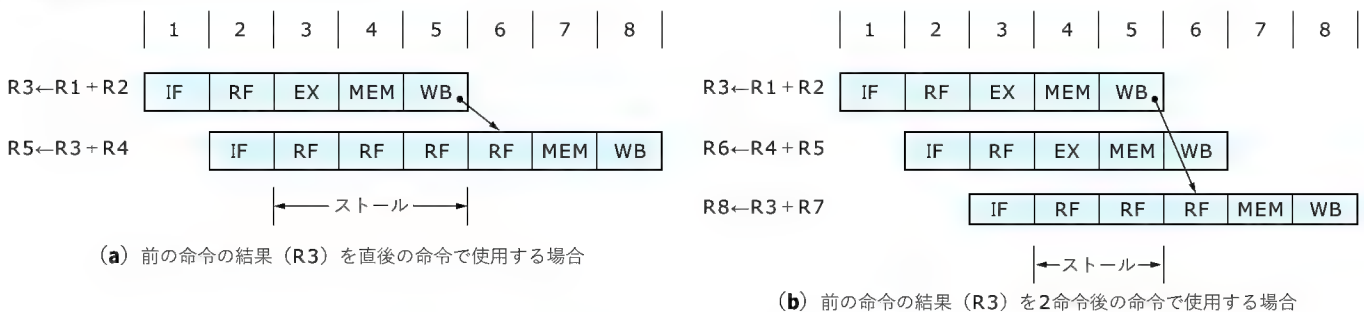
以上はデータに起因するハザードなので, 総称して**データハザード**と呼ばれる. しかし, (2)および(3)のハザードは命令の実行順序が狂わない限り発生しない. 通常のパイプラインでは発生しないが, スーパースカラ構造では発生することがある. これは後述の章で説明する.

当面の課題は(1)の RAW ハザードである. これは, フォワーディング, バイパス, または, ショートサーキットと呼ばれる手法で解決可能である. つまり, EX, MEM, WB ステージから RF ステージへのバイパス回路を設けることで解決できる(**図 4**). RISC では, パイプライン処理を乱さないために, フォワーディングはなかば常識である.

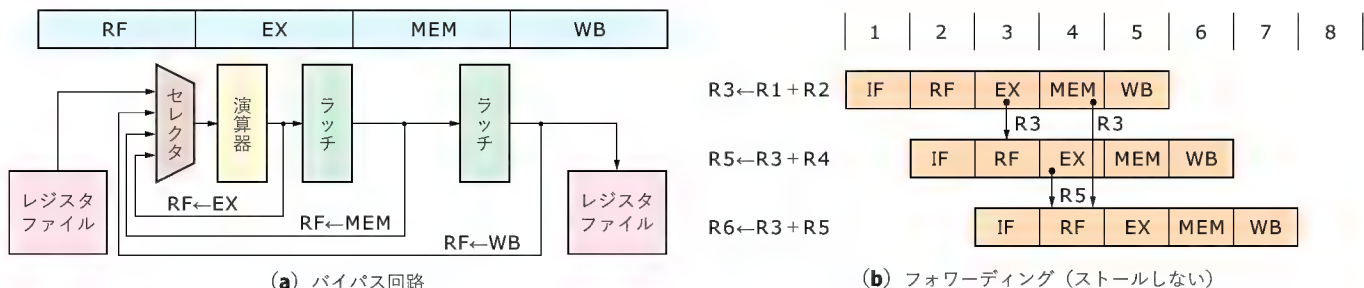
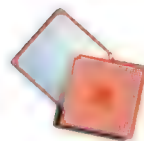
しかし, パイプラインのステージ数が多い場合, 具体的には, レジスタをフェッチするステージ(RF)とレジスタへの書き込みステージ(WB)の間の段数が多いと, 各ステージから RF ス

〔図 3〕RAW(Read After Write)ハザード

(a) 前の命令の結果(R3)を直後の命令で使用する場合

(b) 前の命令の結果(R3)を2命令後の命令で使用する場合

〔図 4〕バイパス回路とフォワーディング

(a) バイパス回路

(b) フォワーディング(ストールしない)

テージへのバイパス経路がその段数分必要なので，実行ステージ(EX)へ与えるデータのセレクタが巨大になってしまう．これはもちろん動作周波数にも影響を与える．どの程度フォワーディングを行うかは悩ましいところである．

● ロード遅延と遅延ロード

ロードした値を直後の命令で使用する場合を考える．この場合，MEMステージで値が初めて確定する．このとき後続命令はEXステージにあるのでフォワーディングは不可能である〔**図5(a)**〕．なにも対処しないと変更前のレジスタの値をフェッチしてしまう．これを**ロード遅延**という．

このため，プログラムの意図どおりに命令を処理するためには，パイプラインの**インタロック**が必要となる．インタロックとはハザードの有無をテストし，ハザードがある場合はハザード原因が解決するまでパイプラインを停止する機構である．

また，停止しているサイクルを**パイプラインストール**(パイプラインバブル)と呼ぶ．**図2**で示す5ステージ構成のパイプラインなら1クロックストールさせればよい〔**図5(b)**〕．

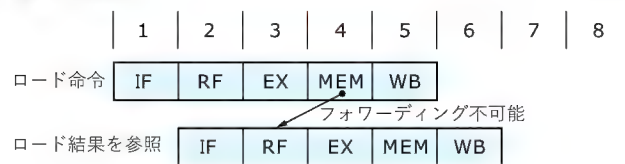
パイプラインのストールは，処理性能の低下を意味する．それを回避する手法の一つは，プログラムの意図を損なわない範囲で命令の順序を入れ替えることである．いまの場合，1クロック分(1命令分)待ち合わせればいいので，ロードした値を参照する命令と後続の無関係な命令を入れ替えればよい．

入れ替えるべき適当な命令がない場合は，NOP命令を挿入することになる〔**図5(c)**〕．この手法はデータハザードの回避にも有効である．このような命令入れ替えや命令挿入を，**命令スケジュール**と呼ぶ．RISCのアセンブラは命令スケジュールを当然のように行っている(禁止の設定も可能)．つまり，アセンブラが「勝手に」最適化するので，プログラマが書いたとおりの順序でコード生成が行われるとは限らないのである．この事実を知ったとき，筆者は少々衝撃を受けたが，いまでは慣れてしまった．

RISCは制御構造の単純化を目標としているから，インタロックは歓迎すべきものではない．ロード遅延をそのまま許し，アセンブラによる命令スケジュールによってのみストールを回避しようという考えがある．これが**遅延ロード**である．MIPSのR2000/R3000は遅延ロードを許すアーキテクチャを採用している．

ただし，R4000からはインタロックするアーキテクチャに変更された．これは，現実問題として，命令の並び替えができる場合が少なく，多くの場合NOP命令が挿入されてしまうからであろう．NOP命令の挿入により，全体としての命令処理は1クロック余分にかかるが，これはストールで1クロックインタロックしても同じである．それならNOP命令がない分，命令コードのサイズを小さくできるという利点がある．

● 制御ハザード

パイプライン処理を乱すハザードにはデータハザードのほかに**制御ハザード**がある．これは分岐によるハザードである．ブランチハザードともいう．RISCでは，条件分岐は汎用レジスタの値で分岐条件を決定する．MPUによっては，CISCと同じく条件フラグを採用しているものもある．この場合の制御ハザードはフラグハザードともいう．

〔**図5**〕**遅延ロード**

さて，条件分岐の場合，分岐条件が確定するまで分岐先の命令フェッチができない〔**図6(a)**〕．これによるストールは命令スケジュールで回避することはできず，性能に与える影響が大きい．

条件フラグを使用する場合でも命令スケジュール可能だが，そのアルゴリズムは非常に難しい．RISCが条件フラグを採用しない理由の一つがコンパイラでの命令スケジュールを簡単にするためである．なお，条件分岐で分岐条件が成立して分岐することをTAKEN，分岐条件が成立せず分岐しないことをNOT TAKEN(あるいはNO TAKEN)という．

● 遅延分岐

パイプライン処理を乱さないため，ストール期間中も(通常は無効化してしまう)分岐命令の後続命令(これを遅延スロットという)を実行させるという考えがある．**図2**に示すパイプラインでは，EXステージでTAKEN/NOT TAKENが決定される．したがって分岐先の命令フェッチは，1クロックのストール後に実行可能である．

TAKENする場合，通常なら分岐命令の後続命令は実行を禁止しなければならない．しかし，その遅延スロットの命令を実行してから，分岐先の命令をフェッチする構造にすれば，パイプラインはストールする必要はない〔**図6(b)**〕．TAKENしない場合は，もともとストールしない．これを**遅延分岐**と呼ぶ．

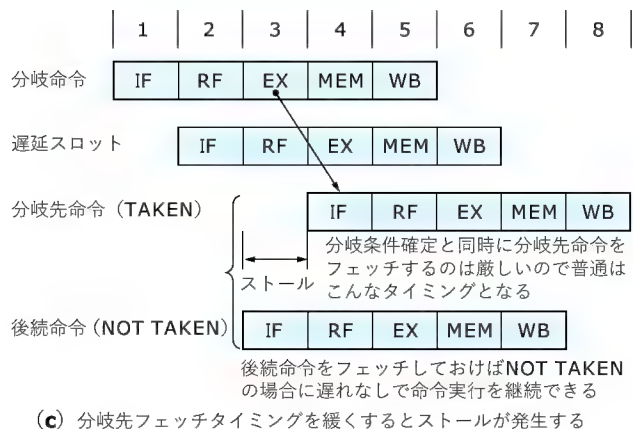
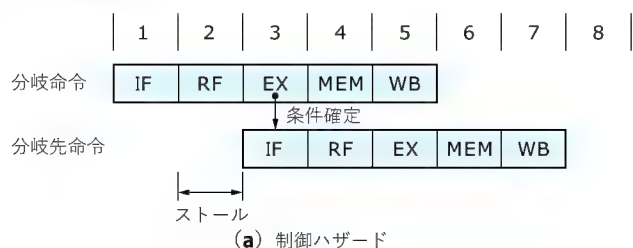
このような遅延スロットを設ければ，命令スケジュールを行うことができる．分岐命令の前方にある命令を遅延スロットにもってくることで，分岐命令によるストールはなくなる．ただし，遅延スロットに入れる適当な命令がない場合は，NOP命令を入れることになる．

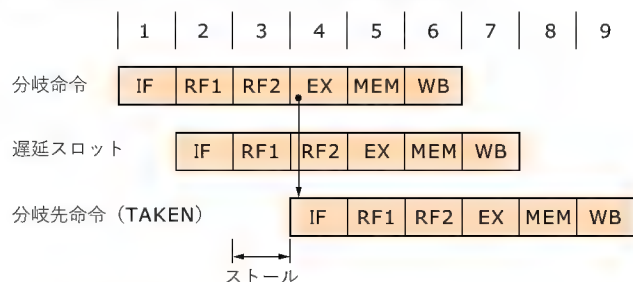
R2000/R3000のパイプラインはこのようになっているが，現実問題としては，分岐命令の分岐先アドレスもEXステージ

〔図6〕遅延分岐

（a）制御ハザード

（b）遅延スロットを設けるとストールがない

（c）分岐先フェッチタイミングを緩くするとストールが発生する

〔図7〕命令デコードが2ステージの場合の制御ハザード

1 2 3 4 5 6 7 8 9
分岐命令 IF RF1 RF2 EX MEM WB
遅延スロット IF RF1 RF2 EX MEM WB
分岐先命令（TAKEN） IF RF1 RF2 EX MEM WB
ストール

で計算される（したがって，分岐条件を判断するための専用の演算器が別個に必要である）ため，それとほぼ同時に分岐先を命令フェッチするのはタイミング的に厳しい．動作周波数を向上させるためには，遅延分岐を採用しつつも，もう1クロック遅れさせるのが望ましい〔**図6**（**c**）〕．このあたりをうまく回避するのが回路設計技術ということもできるが，一般的には，分岐予測を行うことでストールを解消することが可能である．

さて，制御ハザードではTAKENの決定が遅いほどストール期間が長くなる．これはステージ数の多いパイプラインで顕著になる．たとえば，可変長命令を採用するx86のようなMPUにおいては命令デコードに時間がかかる．一般的には，パイプラインで少なくとも2ステージ分が必要である．

たとえば，

IF　RF1　RF2　EX　MEM　WB

の6ステージからなるパイプラインを考える．TAKENの決定はEXステージなので，これまでの説明より1クロック遅いことになる．このとき分岐命令でのストールは2クロックである（**図7**）．1クロックを遅延スロットで埋め合わせるとしても，さらに1クロックだけ処理に余計な時間がかかる．あとで述べるスーパーパイプラインでは，EXステージより前のステージ数がさらに増加し，分岐命令のストールによる性能低下は深刻なものとなる．

● 分岐予測

分岐命令の処理を高速化するために，**分岐予測**という機構が採用される．これは，分岐先アドレスをパイプラインのより早いステージで生成し，分岐先の命令フェッチを早期に行う手法である．具体的には，分岐ターゲットバッファ（BTB：Branch Target Buffer），または分岐予測テーブル（BPT：Branch Prediction Table，BHT：Branch History Table）と呼ばれるキャッシュを用意し，分岐命令のアドレス，分岐履歴情報，予測される分岐先アドレスを格納しておく．

命令フェッチ時（IFステージ）にBTBを参照し，ヒット（登録してある分岐命令のアドレスと命令フェッチアドレスが一致）すれば，分岐履歴情報にしたがって，分岐先アドレスを出力し，命令フェッチを行いながら，TAKEN/NOT TAKENの判定を待つ．予測が成功すればフェッチした命令をそのままデコードすればよい．

予測が失敗すれば，実際にEXステージで計算されるアドレスから命令フェッチをやり直し，BTBの分岐履歴情報を更新する（**図8**）．BTBにヒットし予測が成功する場合はストールがなくなる．BTBにヒットしない場合は，分岐予測を行わない場合と同じタイミングで分岐命令が処理される．

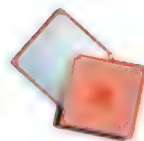
しかし，BTBにヒットするのに予測が失敗する場合は，何もしない場合に比べて，パイプラインの回復処理にかえって時間がかかってしまうことがある．これが，分岐予測失敗時のペナルティである．したがって，分岐予測を採用しても予測が失敗ばかりすると，かえって性能が低下するのでヒット率を向上さ

せるための工夫が必要である．

図8のパイプラインのモデルではBTBにヒットすると予測した分岐先アドレスから命令フェッチを行うが，MPUによっては(予測する)分岐先の数命令をBTBに格納しておき，そこから命令をフェッチする方法を採用する．こうすることにより，パイプラインは予測していない方向の命令も同時にフェッチできるので，分岐予測が失敗した場合のペナルティを最小化できる．

また，分岐予測の成功する確率が高いと思われる場合は，TAKEN/NOT TAKENが決定するまで，予測した分岐先から命令をどんどん先取り(プリフェッチ)する手法もある．パイプラインのステージ数が大きく，TAKEN/NOT TAKENの決定がパイプラインの遅い(後段の)ステージで行われる場合，予測が成功すれば効果的である．逆に予測が失敗したときのペナルティは大きくなる．分岐予測の成功率によほどの自信があるか，失敗時の回復処理がかなり高速化されてないと採れない方式であるが，最近のMPUではけっこうポピュラーである．

● 分岐予測の方法

予測の方法は分岐履歴情報による場合が多い．これは分岐する確率を示す1～2ビットのフラグであり，BTBに登録されている分岐命令ごとに存在する．分岐履歴情報が1ビットの場合は1であるとき「分岐する」，0であるとき「分岐しない」と予測する．これは，その分岐命令が過去1回で分岐したか否かを示している．つまり，以前分岐した分岐命令は今回も分岐すると予測するわけである．

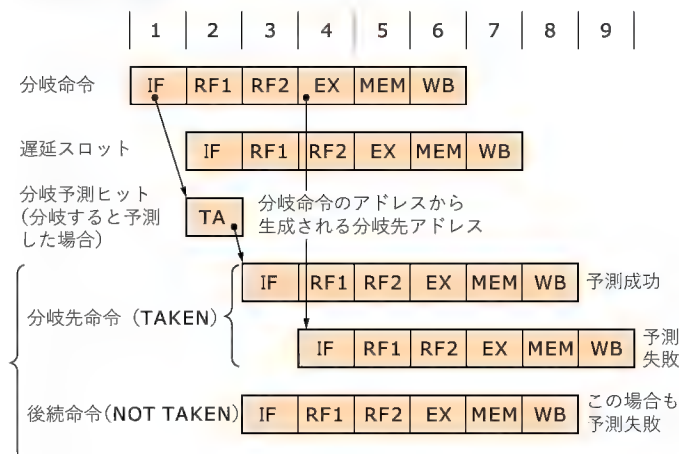
分岐履歴情報が2ビットの場合はもう少し慎重である．ビット列の意味のもたせ方はいろいろ考えられるが，たとえば，11，10で「分岐する」，01，00で「分岐しない」と予測する．これは，その分岐命令が，過去2回において何回「連続して」分岐したかを示す．分岐する傾向が大きい方向に予測するわけだ．

なお，分岐する(と予測する)分岐命令のみをBTBに登録する方法もある．この場合は分岐履歴情報は不要で，BTBにヒットすれば「分岐する」，ヒットしなければ「分岐しない」と予測する．この場合，分岐予測が成功する確率は，分岐履歴情報が1ビットの場合とほぼ同等であるが，BTBの回路規模は約半分になる．

分岐予測を行わない場合で，分岐命令を高速化する方法として，分岐先と分岐元の命令を同時にプリフェッチする手法もある．それに関係する特許は，昔，山のようにあった．これは回路規模が大きくなるため，あまり現実的でない．といいつつも，インテル系のMPU(とくにIA-64)ではそのような説明をよく目にする．ただし，具体的な実装方法は不明である．やはり，特許が絡んでいるようだ．

● 構造ハザード

構造ハザードとは，パイプラインの二つ以上のステージが一つしかないハードウェア資源を取り合うために生じるハザードである．たとえば，5ステージで構成されるパイプラインでは，1時刻に五つすべてのステージが実行される可能性がある．もし，各ステージで同一の演算器などを使用する場合は競合する

〔図8〕分岐予測

ので，優先されるステージ以外は待ち合わせをする必要がある．

RISCの場合，ほとんどのハードウェア資源は競合しないように設計されているのであまり問題はない．しかし，例外もある．それはキャッシュ(あるいはメモリ)である．**図2(b)**をもう一度見ていただきたい．時刻4において命令1のMEMステージと命令4のIFステージが重なっている．もし，命令1がロード/ストア命令であり，命令とデータキャッシュの区別がなく単一のキャッシュしかない場合は，IFステージもMEMステージもキャッシュアクセスなので，資源の競合が生じる．キャッシュが存在しない場合もメモリアクセスの競合が生じる．この場合は，先にある命令1のMEMステージを優先させ，命令4のIFステージをインタロックして待ち合わせることになる．これは，できるだけパイプラインをインタロックさせないというRISCの考え方に反する．

そこで，多くのMPUでは命令とデータを二つのキャッシュに分割して同時にアクセスできるようにしている．これならアクセスの競合によるインタロックは発生しない．このように命令とデータの供給経路を独立にする方式をハーバードアーキテクチャという．

なお，命令とデータに関しては，TLBが一つしかない場合，アドレス変換時にも資源の競合が生じる．それを避けるため，命令用とデータ用のTLBを独立に用意するアーキテクチャもある．多くの場合，命令はアクセスするアドレス範囲が小さい(あるいは連続している)ため，命令用のTLBをマイクロTLBとして，仮想アドレスと物理アドレスのペアを本当のTLBからキャッシュしてもっているのが普通である．

● ステージの処理時間が不均一なパイプライン

さて，パイプラインのステージ間の実行時間が均一でない場合を考える．RISCは命令を1クロックで実行するのが基本であるが，乗除算や浮動小数点演算など1クロックで実行するのが難しい場合もある．

いま，実行ステージ(EX)の処理が4単位時間かかるものと

する(**図9**)．この場合，EXステージが終了するまで同時実行中の他のステージも待ち合わせをするので，パイプラインのスループットは実行ステージの処理時間に依存する．ほかのステージの処理時間は実行ステージの処理時間に隠れてしまう．実行ステージの処理時間が長いだけならまだよい．ほかのステージもまちまちな処理時間を有する場合は悲惨である．不均一であればあるほど，パイプラインの処理時間は各パイプラインステージの処理時間の総和に近づいていく(パイプラインの意味がなくなる)．このため，実行ステージ以外のステージの処理時間を均一にすることが肝要である．

パイプラインにおいて実行(処理)時間がかかるのは，特定命令の「実行ステージ」のほかに，メモリの速度に依存する「命令フェッチステージ」や「オペランドフェッチステージ」がある．RISCは，キャッシュを採用することで命令フェッチやオペランドフェッチの処理時間を1クロックに押し込めようとしている．

典型的なRISCであるMIPSアーキテクチャにおいては，全命令の実行クロックを1クロックとするために，実行時間がかかる乗除算は，通常のパイプラインとは独立して並列実行する．そして，乗除算の結果は汎用レジスタではなく，専用のレジスタに格納される．つまり，乗除算命令では汎用レジスタ間のデータハザードは発生しない．このため，乗除算命令の処理は通常のパイプライン動作に影響を与えない．乗除算が完了した後で，専用レジスタから演算結果を取り出せば(専用レジスタから汎用レジスタへの転送命令が用意されている)インタロックは発生しない．

初期のMIPSプロセッサであるR3000では，乗算と除算の実行時間がそれぞれ12クロックと35クロックである．乗算命令に関していえば，実行を開始してから12クロック後に結果を取り出せばインタロックは発生しない〔**図10(a)**〕．プログラム的には乗算命令と結果を取り出す命令の間が12命令分空いていればよい．

一方，12クロック未満で結果を取り出そうとすると，アーキテクチャ的には不本意ながらインタロックしてしまう〔**図10(b)**〕．現実的には乗除算命令と結果を取り出す命令の間はせいぜい3命令程度しか空けることができないので，乗除算命令があるとほとんどの場合インタロックしてしまうのだが，コンパイラの頑張りによってはインタロックしない可能性を残している．

3 パイプラインを効率良く動かす各種の方法

● 効率的なパイプライン処理が可能になった理由

歴史的に見れば，キャッシュメモリ(高速なローカルメモリ)がまだ高価で外付けのキャッシュすら現実的でなかった時代，プロセッサの処理はメモリからの命令フェッチにいちばん時間がかかっていた．当然の流れとして，プロセッサの性能を上げるためには，フェッチする命令数を減らすこと，1命令で行う処理を増やすことが考えられた．結果として，上述したように実行ステージが長くなる傾向になるのだが，多くの場合はいちばん時間のかかる命令フェッチと，あまり時間のかからない命令のデコードおよび実行をオーバラップ(パイプライン処理)させて実行効率を上げることが可能になる．これが，その時代の最適解であった．そして，それこそがCISCの考え方である．

その後RISCという選択肢が現れてきた背景には，キャッシュが一般的になり，命令フェッチがもはやプログラムの実行に支配的でなくなったことがある．命令のデコードや実行時間が命令フェッチ時間の影に隠れなくなり，実行する命令数よりも1命令の実行時間のほうが性能に対し支配的になった．RISCでは，基本的に1クロック実行なので，CISCに比べて命令実行時間が1/3から1/5になる．1命令が単純な分，同じ処理に要するコード量は増加するが，RISCになることによる命令数の増加はわずか30～50％であるという．差し引き，性能は向上する．

また，RISCでは命令が基本操作に限定されているので，コンパイラによる最適化が行いやすいという利点もある．まあ，現実には，基本的な命令だけで優れた最適化ができるということをMIPSやSPARC用のコンパイラが実証できたためにRISCがメジャーになったともいえるのだが，CISCからRISCの流れは歴史の必然であった．

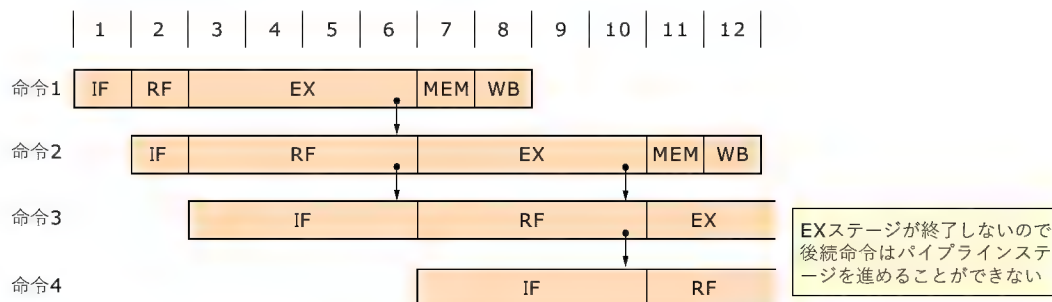
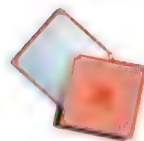
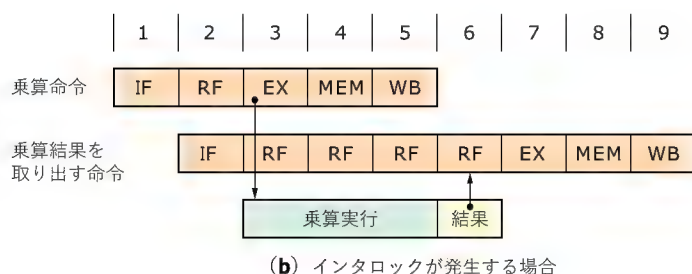
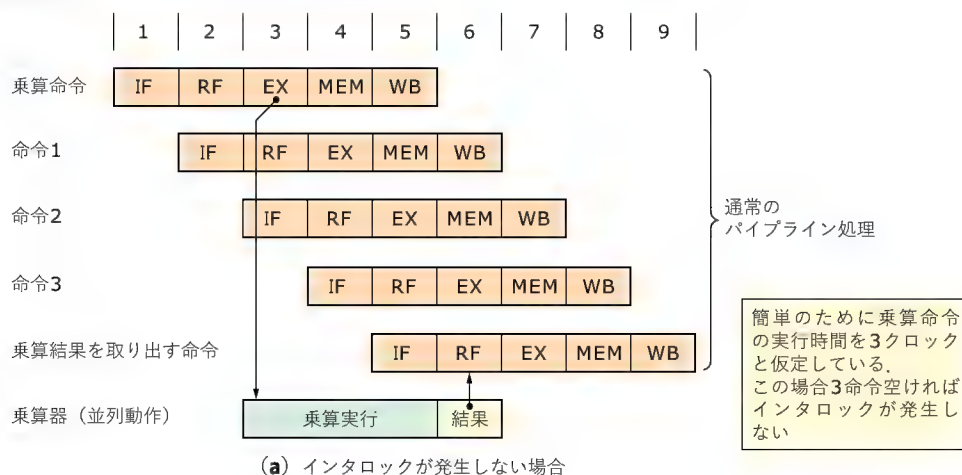
● CPIとMIPS値

パイプライン処理における命令の実行効率を表す指標として，

〔**図9**〕実行ステージが長いパイプライン

〔図10〕MIPSの乗除算命令

(a) インタロックが発生しない場合

(b) インタロックが発生する場合

CPI（Clock cycles Per Instruction）がある．これは1命令を実行するのに必要なクロック数である．RISCの当初の目標は，キャッシュと効率的なパイプライン処理でCPIを1にすることにある．実際，RISCはキャッシュにヒットしパイプラインにインタロックがない場合はCPIが限りなく1に近づく．インテルのMPUの平均CPIに関しては，80386やi486を設計した技術者の一人であるPatrick Gelsingerのレポートがある．それによると，**表1(a)**のような値が出ている．

MPUが進化するにつれてパイプラインの効率が上昇しているのがわかる．さすがインテルというところだろうか．i486でCPIが急激に改善したのは，キャッシュの恩恵といわれている．CISCでありながらRISC並みのパイプライン処理を採用したことも一因であろう．現在のPentiumのCPIは0.6～0.7であるという（ちょっと性能が良すぎる感もあるが）．これは後述の章で説明するスーパースカラの恩恵である．

CPIはMIPS（Million Instructions Per Second）値と密接な関係がある．MIPS値とは1秒間に実行できる命令数（100万命令単位）だから，動作周波数とCPIが決まれば，

周波数（MHz単位）÷ CPI

という計算式で，MIPS値が求まる．この式で，上のx86プロセッサのMIPS値を計算すると**表1(b)**のようになる．

実際に公表されるMIPS値は，Dhrystone MIPS（最近では

〔表1〕x86系CPUのCPIとMIPS値

CPU	CPI
8086	15.0
80286	6.0
80386	4.5
i486	1.7

(a) x86系CPUのCPI

CPU	MHz	MIPS
8086	5	0.33
80286	8	1.33
80386	12	2.67
i486	25	14.71
Pentium	66	110

(b) x86系CPUのMIPS値

DMIPSと略記されることもある）なので，もう少し高い値になっているかもしれない．これは，Dhrystoneベンチマークを実行した性能が，1MIPS相当のVAX-11/780の何倍であるかを表すものである．

Dhrystone MIPSでは，コンパイラの性能しだいで，シングルパイプラインのMPUのCPIが1を割ることも多く，直感的ではない．しかし，現在実際に使用されているMIPS値はDhrystone MIPSが主流なので，慣れが必要である．

もっとも最近のx86系は，MIPS値の公表をやめてしまっている（表向きの理由はいろいろあるが，発表するとCPIの大きさが問題となるからだろう）ので，性能を比較するには動作周波数に頼るしかない．各メーカーは独自の基準で従来品との相対性能を公表しているが，異なるメーカー間での性能比較はできない．いくら動作周波数が高くてもCPIが悪ければ何にもな

らないのだが，メーカーやマスコミはこの点を意図的にうやむやにしているようにも思える．

インテルはPentium4で3GHz以上の動作周波数を実現した．実効性能はPentium4と同等であるが，動作周波数ではPentium4に劣るAthlonXPを有するAMDは，周波数の大きさによる優位性のアピールから実効性能の優位性のアピール（モデル番号の採用）に方針転換した．

● スーパーパイプライン

MPUを高い周波数で動作させるためには，パイプラインの1ステージあたりで実行する論理を減少させる必要がある．単純に考えると従来1ステージで実行していた処理を2ステージに分割することである．つまり，高速な動作周波数になるにつれてパイプラインのステージ数が増加する傾向にある．いま，パイプラインのステージを

IF1　IF2　RF1　RF2　EX1　EX2　MEM1　MEM2　WB1　WB2

としてCPIを試算してみよう．**図11**(**a**)では4命令を8クロックで実行しているのでCPIは2.0である．一方，**図11**(**b**)では4命令を13クロックで実行しているのでCPIは3.25である．スーパーパイプライン構成にすることでCPIは約1.5倍に増加してしまう．しかし，動作周波数を2倍に引き上げることができれば実質的な性能は向上する．これがスーパーパイプラインの考え方である．

スーパーパイプラインを最初に採用したのはMIPSのR4000である．これは当初100MHz動作であったが，最終的には250MHz動作を達成している．ほぼ同時期に登場したDECのAlpha（21064）は200MHz動作を達成していた．これは1990年代の始めとしては驚異的な動作周波数だった．このためAlphaは，世界最高速のMPUとしてギネスブックに登録された．

最近では，

動作周波数を上げる＝パイプラインのステージ数を増やす

という図式が常識のように語られるようになった．インテルのPentium4（Willamette）は20ステージのパイプライン構成で2GHz以上の動作を目指した．そして，Northwoodで3GHzを超えた．IPコアの分野でも，Lexra社がLX4189（MIPS系）でパイプラインを従来の5ステージから6ステージに変更することで，初めて250MHz以上の動作周波数を達成したと発表した．

動作周波数を上げるためにはパイプラインのステージ数を増やす必要があるのは本当だが，逆は必ずしも真ではないと思うので，そんな単純なものではない．しかし，これからのMPU設計においては，パイプラインのステージ数を増加して動作周波数をかせぎ，それによるCPIの増加は分岐予測を高度にすることで補っていく傾向になるのは間違いないだろう．

● プリフェッチとデカップル（decouple）構成

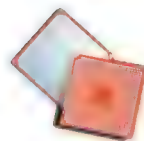
命令フェッチが命令キャッシュにヒットする限りは，各サイクルごとに命令デコーダに命令が滞りなく供給されるので，プリフェッチをして命令をFIFOなどに蓄えておく必要はない．しかし，命令キャッシュミスが発生すると命令供給が停止するので，パイプラインがストールしてしまう．それを防ぐためにプリフェッチは有効である．命令デコード以降のパイプライン処理とは，独立に命令を絶えずプリフェッチしておけば命令デコードにおいて命令の供給が停止する頻度は少なくなる．

命令キャッシュのミスが発生した場合，命令キャッシュへの書き込みと同時にデコーダへ命令をバイパスする「命令ストリーミング」もパイプラインのストールを低減させる方法の一つである．しかし，命令ストリーミングでは，（通常は）パイプラインクロックよりも遅いバスクロックに同期して命令供給が行われるので，命令ストリーミング中の命令処理はバスクロック同期に近くなり，効率があまりよくない．プリフェッチは，命令キャッシュミスの発生が契機となるわけではなく，無条件に命令フェッチを行っていくので，命令ストリーミングよりも効率がいい（はずである）．

シングルパイプラインではあまりお目にかからないが，**デカップル方式**という構成がある．これは，プリフェッチとよく似た概念であるが，命令デコードと実行ステージの中間にFIFOを置いて，そのFIFOに絶えずデコード済みの命令を格

〔**図11**〕**スーパーパイプラインの効果**

(**a**) 通常のパイプライン

(**b**) スーパーパイプライン

納しておく．こうすることで，FIFO内の命令はソースオペランドが有効である限り，各サイクルごとに命令実行を開始することが可能になる．つまり，オペランドの依存性による命令デコードステージでのストールが緩衝されて見えなくなる（パイプライン効率が上がる）．

当然，命令フェッチとデコードまでのステージと実行ステージ以降は別のクロックに同期して独立に動く．パイプラインがデコードまでと実行以降が分離（decouple）されていることで，デカップル方式と呼ばれる．

デカップル方式の利点は，単純なプリフェッチとは異なり，命令デコードを行うので分岐命令を認識することが可能であり，分岐予測をしながら投機的（speculative）に命令のプリフェッチを行うことができる点である．単なるプリフェッチであれば，分岐する分岐命令以降にある命令をむだにプリフェッチする恐れがある．分岐予測にしたがってプリフェッチを行うことができれば，（分岐予測が当たる限り）命令フェッチのロスはなくなる．

〔図12〕命令書き換えのタイミング

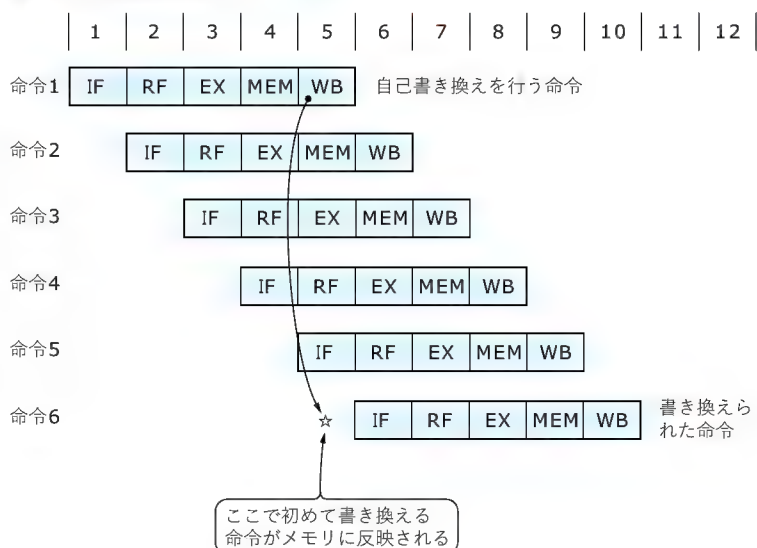
このため，デカップル方式では，分岐予測が有効に働けば，パイプライン処理の中で，命令フェッチと命令デコードステージを無視することができる．たとえば，5ステージのパイプライン処理ならば，2ステージ少ない，3ステージのパイプラインと同等の効率で命令を処理できる．

プリフェッチやデカップル方式での投機的なデコードは，実行ステージ以降で発生するパイプラインストールの合間を縫って行われる．実行ステージ以降でストールがまったく発生しなければ，プリフェッチ機構自身が無意味なものになってしまう．パイプライン効率は落ちないが，プリフェッチをしてもしなくても同じ効率にしかならないので，余分な回路ということになる．

実際問題として，シングルパイプラインではロード遅延とデータキャッシュミス以外では実行ステージ以降でのストールは発生せず，プリフェッチは，その回路規模の割には，性能は向上しないと思われる．しかし，2命令以上を同時に処理するスーパースカラにおいては，命令デコードの倍以上の速度で命令が処理されていくので，プリフェッチや投機的デコードの機構を用意しておかなければ命令供給が命令消費に追いつかなくなる．デカップル構造においてはスーパースカラの解説の章で詳しく説明する．

● 自己書き換えとパイプライン

昔，8086や68000というMPUが全盛だった頃，プログラムのコードサイズを削減するために，命令コード領域をストア命令で書き換えて実行するという技が重宝されていた．これは**自己書き換え**と呼ばれる手法である．

自己書き換えは，パイプラインを採用するMPUでは期待どおりの動作をするとは限らない．それは，パイプラインのステージを考えれば明らかで，書き換えた命令のフェッチ（IF）は書き換える命令のライト（WB）以降でなければならないためである．たとえば，

IF　RF　EX　MEM　WB

という5ステージ構成では，最低5命令以後を書き換えなければ，そこを正しくフェッチできない（**図12**）．また，命令のプリフェッチを行う場合は，一概に何命令後を書き換えれば大丈夫かということは保証できない．書き換えた場所にジャンプすればよいという考えもある．この方法も，分岐予測などで命令フェッチが先行する場合は，うまくいかないことがある．

ところで最近のMPUは，命令キャッシュとデータキャッシュが分離されているので，単純に命令コードを書き換えることはできない．ストア命令を実行してもデータキャッシュの内容が変更されるだけで，命令キャッシュの内容は変わらないからである．

ただし，（OSに限られるが）特権命令を使えば，書き換えたアドレスに対応する命令キャッシュの内容を無効化することで，自己書き換えを実現できる．もし，ライトバックキャッシュ構成ならデータキャッシュを最初に強制的にライトバックさせることも必要である．

……と，自己書き換えを推奨するような説明をしてみたが，最近のプログラミングではこの技法は好ましくないものとされている．現在ではMMUが内蔵され，十分大きなアドレス空間を使ってプログラムを作ることが可能なので，わざわざプログラムの流れを分かりにくくする自己書き換えを行う理由はない．

とはいえ，仮想記憶のデマンドページングで行われるスワップインは壮大な自己書き換えではないかと考えると，OSなら自己書き換えをしてもいいのかという話も出てくる．リアルタイムOSなどのプログラムのダイナミックリンクも，自己書き換えに近い．

もっとも，有限オートマトンとしてのコンピュータを考えれ

Column

ウェーブパイプライン

パイプライン処理は，各ステージの処理を，通常，一つのクロックに同期させて進めていく．しかし，各ステージの純粋な処理時間は論理の複雑さに依存し，クロックで既定される時間ぎりぎりかかる場合もあれば，クロックで既定される時間より早く終わる場合もある．

クロックのサイクルより短い時間でパイプラインのあるステージが終了する場合，その空き時間をむだにしないような実装ができれば，相対的に処理時間を短縮することができ，見かけ上のクロック周波数を向上することができる．

図Aを見てほしい．IF，RF，EX，DC，WBからなる5ステージのパイプラインを考える（MEMステージはデータキャッシュアクセスなのでDCとしている）．図に示すように，各ステージの処理時間を仮定する．ここでは，IFとDCがキャッシュアクセスで，もっとも遅いステージになっている．RFがデコードステージでその次に遅い．EXとWBはレジスタアクセスなので比較的高速である．

従来のパイプライン方式では，**図A(a)**のように5クロックかけて1命令の処理が終了する．これを，ステージの空き時間を詰めていくと，**図A(b)**のように，4クロック程度で1命令の処理が終了する．処理時間が4/5になったのだから，同じ処理をする場合のスピードは1.25倍になる．つまり，見かけ上のクロック周波数は1.25倍になる．

しかし，MPUの動作はクロック同期が基本であるが，各ステージを駆動するクロックの周期（周波数）さえ一致していれば，各ステージを同じタイミングで処理しなくても，安定なパイプライン処理を維持することができる．単純には，ステージごとに独立なクロックを用意することが考えられる．**図B**ではパイプラインのステージ数と同じ5相クロックを用いたパイプラインを示す．

この場合，5系統のクロックの周波数は同一であり，IFステージはCLK1，RFステージはCLK2，EXステージはCLK3，DCステージはCLK4，WBステージはCLK5に同期して動作している．このように多相クロックを用いることで，見かけ上のクロック周波数を向上できる．このようなパイプライン構造を**ウェーブパイプライン**（wave pipeline＝波打ったパイプライン）と呼ぶ．あるいは，最大レートパイプライン（maximum rate pipeline）と呼ばれ，文字どおり，パイプラインの処理速度を最大限に上げることができる．

ウェーブパイプラインを行うためには制限がある．各ステージの

〔図A〕ウェーブパイプライン

(a) 単相クロック同期

(b) クロックにしばられない場合

ば，自己書き換えができるのは当然の機能/属性である．プログラムで行う自己書き換えとOSのページングは粒度（書き換えから実行までの時間的空間的距離）の大きさの違いとして説明される．つまり，粒度の小さい自己書き換えは推奨されないということだ．

まとめ

シングルパイプラインの概要について説明してきた．思えばヘネパタは偉大だった．いうまでもないがHennessyはMIPS RISCの生みの親，Pattersonはバークレー RISCの生みの親である．この2人によって著わされたヘネパタは，日本のRISCメーカーの技術者に多大な影響を与えた．日立のSHシリーズやNECのV800シリーズはヘネパタで示されたアーキテクチャ（とくにパイプライン）を参考にしているといわれている．それだけDLX（R3000）のパイプラインが洗練されているということなのであろう．パイプラインの実例も，はからずもMIPSアーキテクチャの例が多くなってしまった．なおヘネパタは，日本語訳も出ているので，未読の方は一度は目を通すことをおすすめする．

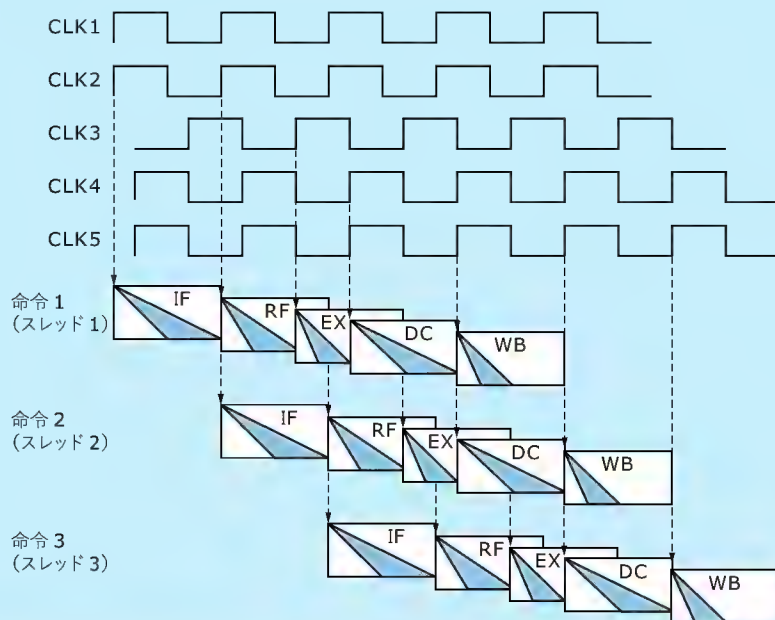
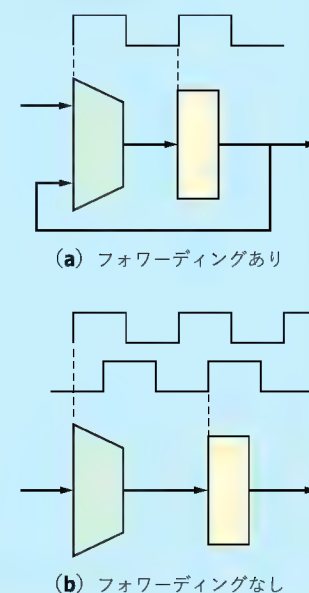
次章では，実際の各種プロセッサにおけるパイプラインの構造について解説する．

なかもり・あきら　フリーライタ

クロックが一致していない(ずれがある)ので，WBステージの前の結果をRFステージにフォワーディングすることが難しい(**図C**)．フォワーディングができなくなると，見かけ上の動作周波数は向上しても，ハザードによりCPIが増加してしまう．ウェーブパイプラインは，ハザードが発生しにくい状況下で効果を発揮する．

ハザードが発生しにくい場合とは，どういう状況であろうか．これは，パイプラインに順次投入されていく命令間に依存性のない場合である．この例としてすぐに思いつくのは，多数のスレッドをパイプライン処理する場合である．スレッドとはプログラムのうちで並列実行できる部分を抽出したもの(ゆえに，多くの場合，依存性はない)である．たとえば，近年はやりのマルチメディアアプリケーションはスレッドに分解するのに適している．このため，ウェーブパイプラインは多くの場合，マルチスレッドの処理に適用される．

ウェーブパイプラインは，大学レベルでは多くの研究がなされているが，商用のものはまだ存在しない．これは多相クロックを使用するため，クロックの遅延を合わせ込むのが難しく，遅延解析に向かないためである．つまり，安定量産できるための回路設計は難しい．さらに，多相クロックの場合，論理合成が難しく，手作業による専用回路設計が必要になる．それにかける工数に比して利益の見込みが少ないので，企業が実践するには苦しいものがある．

〔図B〕5相クロックを用いたウェーブパイプライン

〔図C〕演算後のフォワーディング

(a) フォワーディングあり

(b) フォワーディングなし

実際のプロセッサはどのように動いているか

パイプライン処理の実際

中森　章

RISC のパイプライン処理は，見事なまでにヘネシー＆パターソンが提唱した 5 ステージのパイプラインにしたがっている．MPU によっては，5 ステージでない場合（部分的あるいは全面的なスーパーパイプライン）もあるが，基本は同じである．もっとも，これはシングルパイプラインの場合で，スーパースカラの場合は少し事情が異なる．ここではシングルパイプラインの代表ともいえる，R3000，SH シリーズ，ARM，V800 シリーズのパイプライン構造を解説する．（筆者）

1 R3000 のパイプライン

● RISC の基本そのままのパイプライン

R3000 のパイプラインは，基本的には前章の図 2 で示した RISC のパイプラインと同じである．IF，RF，EX，MEM，WB の 5 ステージで構成される．実際には φ 1，φ 2 の 2 相クロックで動作し，1 クロック間に 2 ステップの処理を行っている．**図 1** に R3000 のパイプラインの詳細を示す．各ステージでの動作は，次のようになっている．

1）IF φ 1

マイクロ TLB（ITLB）を使用して命令の仮想アドレス（IVA）を物理アドレスに変換する．分岐先アドレスは RF ステージの φ 2 で計算され，EX ステージの φ 1 でアドレス変換される．

2）IF φ 2

物理アドレスを命令キャッシュに転送し，命令キャッシュをアクセスする（ICache）．

3）RF φ 1

命令キャッシュのヒット/ミスがチェックされ，命令キャッシュから命令を読み出す（ICache）．

4）RF φ 2

命令をデコードする（ID）．分岐命令の場合は分岐先アドレスを計算する．レジスタファイルをリードする（RF）．

5）EX φ 1 ＋ φ 2

オペランドを他のパイプラインステージからバイパスし，演算する（ALU）．ストアするデータがあれば位置合わせを行う．

6）EX φ 1

分岐命令なら TAKEN/NOT TAKEN を決定する．ロード/ストア命令ならオペランドの仮想アドレスを計算する（DVA）．

7）EX φ 2

ロード/ストア命令なら TLB を使用してオペランドの仮想アドレスを物理アドレスに変換する（DTLB）．

8）MEM φ 1

ロード/ストア命令なら物理アドレスをデータキャッシュに転送し，データキャッシュをアクセスする（DCache）．

9）MEM φ 2

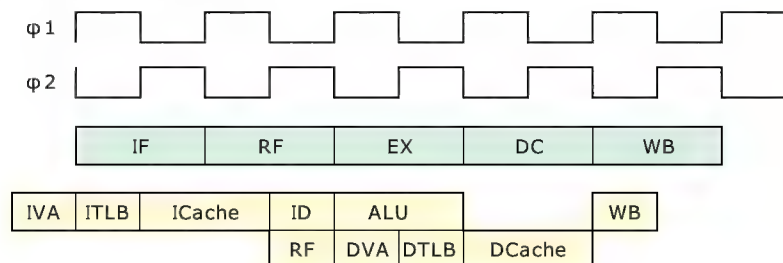
データキャッシュのヒット/ミスがチェックされ，命令キャッシュからオペランドを読み出す（DCache）．

10）WB φ 1

EX ステージでの演算結果をレジスタファイルに書き込む（WB）．ストア命令の場合はデータキャッシュに書き込む．

● 単相クロック動作の R3000

後年，R3000 相当の IP コアを提供する目的で，パイプラインを見直した 4Kc（Jade），4KEc（Emerald），5Kc（Opal）など（これらはコアの名称）は，単相クロック同期に変更しているが，基本的なパイプライン構造に変更はない．**図 2** に R3000 と Jade のパ

〔図 1〕R3000 のパイプライン

IVA : Instruction Virtual Address Generation
ITLB : Instruction TLB Access (Address Translation)
ICache : Instruction Cache Access
ID : Instruction Decode
RF : Register File Access
ALU : ALU Operation
DVA : Data Virtual Address Generation
DTLB : Data TLB Access (Address Translation)
DCache : Data Cache Access
WB : Writeback

イプラインの比較を示す．R3000のパイプラインが図1と一部異なっているが，図1は説明用に簡略化したもので，図2は現実に近いものと理解すればいいだろう．

図2に示すように，単相クロックでの再設計を考えた場合，R3000は多くのクリティカルな操作（命令キャッシュアクセス，レジスタリード，データTLB参照）をパイプラインクロックの立ち下りエッジに同期して行っている．また，データキャッシュアクセスはクロックの立ち上がり同期であり，データキャッシュからリードしたデータの位置合わせ（図2のLA）を同じパイプラインステージ内で行うので，タイミングはかなり厳しい．命令のアドレス計算も，命令キャッシュアクセスの前後に，二つの1/2サイクルのアクセス（IA1，IA2）に分割して行われるので，制御が複雑になる．これらが，IPコアとして容易に論理合成を行うためのボトルネックになっている．また，SRAM（キャッシュ）のアクセスタイミングも厳しく，キャッシュをメモリコンパイラなどで自動生成するのが困難である．

〔図2〕R3000とJadeのパイプライン比較

このため，Jadeではパイプラインが再設計された．具体的には，すべての操作を1フェーズ早めてクロックの立ち上がり同期にした．さらに，命令TLBアクセスとデータキャッシュアクセスを1ステージ早くして，リードデータの位置合わせをキャッシュアクセスと別のステージにもっていった．結果として，すべてのクリティカルな操作は立ち上がりエッジ同期になった．命令のアドレス計算は，命令キャッシュアクセス後の，一つのパイプラインステージに統合された．これらの結果，データキャッシュアクセスのタイミングに余裕ができた．

図2からわかるように，レジスタファイルへのライトを位置合わせの直後（立ち下がり同期）にすることで，パイプラインステージ数を5ステージから4ステージにすることも可能である．しかし，Jadeではクロックの立ち上がり同期にこだわり，結果として5ステージのパイプラインとなっている．

● Jadeパイプラインの利点

Jadeパイプラインは三つの利点があるといわれている．一つ目は，キャッシュアクセスに余裕があること．二つ目は，クリティカルな操作がすべて立ち上がり同期になっているので，ある論理ブロックをユーザーが設計した論理に置き換えることが容易なことである．三つ目は，論理合成ツールによる遅延の調整が容易になることである．本来，論理合成を想定した機能設計は，クロック遅延のばらつき（クロックスキュー）を一定値内に収める操作を容易にするために，クロックの立ち上がりエッジのみを使用する．これを実践したわけだ．

MIPSの発表によると，0.25μmプロセスで製造した場合の動作周波数は，最悪の場合（単純な論理合成）で100～150MHz，典型的な場合（専用設計）で150～255MHzだそうである．クリティカルな操作を立ち上がり同期にしたとはいえ，パイプライン効率はR3000のそれと大差がないのも事実で，この動作周波数が可能なのか否かは実際に回路設計した人にしかわからないだろう．

MIPSはJadeの拡張版の4KEc（コードネームEmerald）でMIPS16命令セットに対応すると発表した．しかし，その実装方法たるや，1段のデコードステージ内でMIPS16からMIPS32へのコード変換を行ってデコードするという，非常に厳しいタイミングを提唱している．Jade開発当時の理念はどこへ行ってしまったのだろうか．もっとも，4Kcと4KEcの構造的な違いは，低消費電力を実現するゲーテッドクロックを行うか行わないかの違いだけである．

なお，先頃発表されたPSP（PlayStation Portable）のCPUコアは4Kcまたは4KEcであって，R4000系ではない．

● Opalのパイプライン

Opalではさらにパイプラインが変更された．Opalのパイプラインは IF，Dispatch，RF，EX，MEM，WBの6ステージで構成され，φ1，φ2の2相クロックで動作するとされている．しかし，論理合成を容易にするために単相クロックを採用しつつも説明上の方便でφ1，φ2を使用しているのではないかと思われる．

Opal自身はスカラプロセッサだが，スーパースカラへの移行の可能性を残している．つまり，ディスパッチステージが命令フェッチとレジスタリード/命令デコードステージの間に挿入

された．このためパイプラインは，Jadeより1ステージ多い6ステージとなる．これは，将来的には，複数の演算ユニットに命令をディスパッチ(発行)するために使用する．命令デコード自体にも余裕ができるので，動作周波数が少し向上する．また，この追加ステージはMIPS16のためのプリデコードステージとしても利用できる．

パイプラインのステージ数が増加することで分岐の性能が悪くなるが，Opalでは静的な分岐予測と命令プリフェッチで対応している．分岐はすべてTAKENするものと仮定し，投機的に6命令をフェッチできる．分岐予測が外れた場合のペナルティは1サイクルにすぎないという(筆者としては懐疑的)．Opalのパイプラインの詳細を以下に示す．

1）IF φ1＋φ2

命令キャッシュにアクセスする(ICache)．命令の仮想アドレスはDispatchステージ(IA1)とRFステージ(IA2)で計算される．

2）IF φ2

マイクロTLBにアクセスし，命令の仮想アドレスを物理アドレスに変換する(ITLB)．

3）Dispatch φ1＋φ2

命令キャッシュのヒット/ミスをチェックする(WS：Way Select)．スーパースカラ構成を採るための命令ディスパッチ用のタイミングを提供する(Dispatch)．MIPS16をサポートする場合のプリデコードタイミングを提供する(MIPS16)．次の命令のための命令の仮想アドレスを用意する(IA1)．

4）RF φ1＋φ2

レジスタをフェッチする(RF)．命令をデコードする(ID)．

5）RF φ1

分岐先の仮想アドレスを計算する(IA2)．

6）EX φ1＋φ2

演算を行う(ALU)．

7）EX φ1

ロード/ストア命令のオペランドアドレスを計算する(DA)．

8）EX φ2

データキャッシュへのアクセス(DCache)．1段目．

9）MEM φ1

オペランドの仮想アドレスを物理アドレスに変換する(DTLB)．データキャッシュへのアクセス(DCache)．2段目．

10）MEM φ2

データキャッシュのヒット/ミスをチェックする(WS)．データキャッシュからフェッチしたデータ，データキャッシュにストアするデータの位置合わせをする(LA)．

11）WB φ1＋φ2

EXステージでの演算結果をレジスタファイルに書き込む(WB)．ストア命令の場合はデータキャッシュに書き込む．

● JadeとOpalの性能……どちらが高い？

MIPSの発表によると，Opalを0.15μmプロセスで製造した場合の動作周波数は450MHz，0.18μmプロセスでは375MHzだそうである．Opalでは，Jadeでわざわざ立ち上がり同期に揃えたデータキャッシュアクセスが立ち下がり同期に変更されていることもあり，スーパーパイプライン構造も採用していないので，本当にこんな高周波数で動作可能かは不明である．

さらにMIPSの発表によると，JadeとOpalの性能(MIPS/MHz)はどちらも，Dhrystone MIPSで1.2であるという．これはR3000とほぼ同じ性能である．Opalに関しては，パイプラインのステージ数が増えているのに，Jadeと同じ性能というのは納得がいかない．4KEcの発表ではMIPS/MHzを1.7としていた．Jadeと実体は同じなのに，この性能アップの理由は不明である．そもそも1.7という値はスーパースカラでないと実現できない．

それは置いておくとしても，同じ性能のIPコアが二つも必要なのかという疑問が残る．MIPSの弁明では，Dhrystoneベンチマークでは真の性能はわからない，実際のアプリケーションではOpalはJadeの2倍の性能があるという．これはキャッシュ容量を2倍にできる点と，64ビット演算と32ビット演算の差と説明されているが……．

JadeにしろOpalにしろ，論理合成可能なRTL(Register Transfer Level)記述で提供されるのだが，目標動作周波数が達成できるか否かは，LSI製造メーカーの技術力によると思う．しかしMIPS社の説明では，JadeにしろOpalにしろ，何も特別なことを行っているわけではなく，誰が作っても200MHz以上の性能は保証するとしている．また，Dhrystoneベンチマークの値が異様に高い理由としては，アーキテクチャをDhrystoneに特化しているらしい．実アプリケーションではあまり効果がないが，Dhrystone MIPSの値は採用の決め手になることが多いので，あえてそのような構造にしているという．

2 SH-1/SH-2/SH-3，そしてSH-5

● 16ビット固定長命令RISC

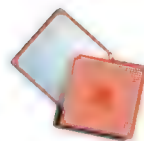
SHシリーズは日立製作所が1992年に発売した，組み込み用途をねらったRISC型32ビットマイクロコントローラとして誕生した．その後，積和演算やMMUを内蔵し，MPUとしての地位を確実にしている．多くのRISCが32ビットの固定長命令であるのに対して，SHシリーズは16ビット固定長命令を採用しコードサイズの削減を図っている．

SHシリーズはDLX(R3000)のパイプラインを参考にしているといわれるが，命令によってパイプラインのステージ数が異なっている点で，CISCの考え方を引きずっているようにも思える．

パイプラインは，次の5ステージから構成される．ステージ構成だけを見れば，R3000と同一である．また，遅延分岐は採用しているが，遅延ロードは採用せずデータハザードが生じる場合はインタロックする．

1）IF：命令フェッチ

2）ID：命令デコード

3）EX：命令実行

4）MA：メモリアクセス

5）WB：ライトバック

IF，ID，EXの3ステージはすべての命令に存在するが，命令によっては，MA，WBステージがない場合もある．おもなパイプラインを**図3**に示す．図を見るとわかるが，パイプラインはSH-1/SH-2とSH-3で少し異なっている．レジスタ-レジスタ間演算（転送を含む）は，SH-1/SH-2ではIF，ID，EXの3ステージで構成されるが，SH-3ではデータを保持するだけのMAステージと，レジスタへ値をライトするためのWBステージが追加されて5ステージ構成になっている．

これらの命令において，SH-1/SH-2/SH-3とも，レジスタのリードはEXステージで行っているようである．そして，演算を行った結果は，SH-1/SH-2ではEXステージのうちに，SH-3ではWBステージでレジスタにライトするようである．

R3000のパイプラインを参考にした（といわれる）わりには，EXステージまでレジスタリードを遅延させたり，SH-1/SH-2では演算結果をEXステージでレジスタにライトさせたりするなど，タイミング的に厳しい設計になっている．これは，レジスタのフォワーディングをまったく行っていないか，フォワーディングの論理を軽くするためと推測される．もっとも，SH-4ではIDステージでレジスタをリードするようになったようで，試行錯誤の痕跡が認められる．

さて，ロード命令はパイプラインの5ステージすべてを使う．WBステージは，最初はロードしたデータをレジスタにライトするためだけに存在していたようだ．SH-3ではレジスタ-レジスタ演算にも適用された．一方，ストア命令はレジスタへのライトがないのでWBステージが存在しない．いずれにせよ，命令の種類に応じてパイプラインのステージ数を可変にするのはCISCの発想である．実質的にはパイプラインのスループットは，最大のステージ数に支配されるのであまり効果はない．

SH-3ではそのことに気づいてか，パイプラインがほとんどの命令で5ステージ固定に改善されたが，ストア命令がなぜ4ステージのままなのかは謎である．

● SH-5

SH-4では性能向上のためにスーパースカラ構成を採用したが，1999年に発表されたSH-5ではシングルパイプラインに戻した．400MHz動作を達成するためには，スーパースカラの制御の複雑さがスピード上のネックになるという理由からだ．

SH-5のパイプラインは，

1）Fetch-1（F1）：命令フェッチ

2）Fetch-Decode（FD）：命令フェッチ＆デコード

3）Decode（D）：デコード

4）Execute-1（E1）：実行

5）Execute-2（E2）：実行

6）Execute-3（E3）：実行

7）Writeback（W）：ライトバック

の7ステージで構成される．

〔図3〕SHのパイプライン

レジスタ-レジスタ間演算 SH-1/SH-2	IF	ID	EX				
SH-3	IF	ID	EX	MA	WB		
ロード命令 SH-1/SH-2/SH-3	IF	ID	EX	MA	WB		
ストア命令 SH-1/SH-2/SH-3	IF	ID	EX	MA			
SH-4のパイプライン（参考）	I	D	EX	MA	S		
SH-5のパイプライン	F1	FD	D	E1	E2	E3	W

E1，E2，E3ステージからDステージへのフォワーディングが可能だというが，だったらSH-4までは「フォワーディングを行っていなかったのか」と突っ込みたくなるが，RISCであるからには，そんなはずはないだろう．

パイプラインのステージ数の増加にともなう分岐命令の性能低下を補うため，SH-5ではSplit Branch（分割分岐とでも訳すか？）という方式を採用している．これは，分岐先アドレスを計算して命令プリフェッチを行っておき，実際の分岐命令でその命令を実行するという2段階の構造で分岐命令処理を実現する．そのためにPTA（Prepare Target Address）という命令が用意された．

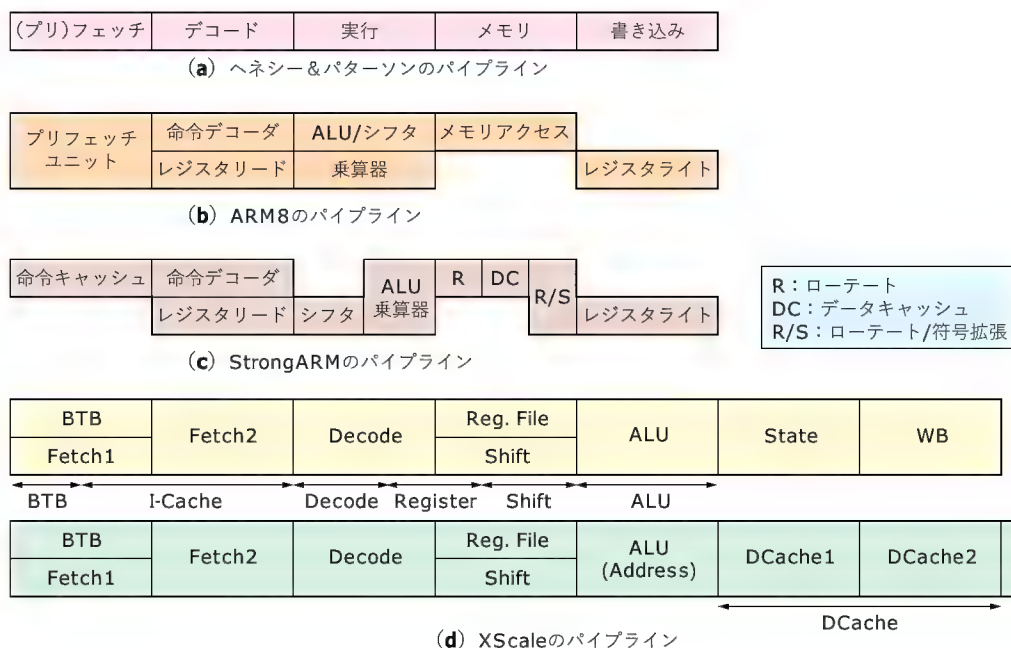
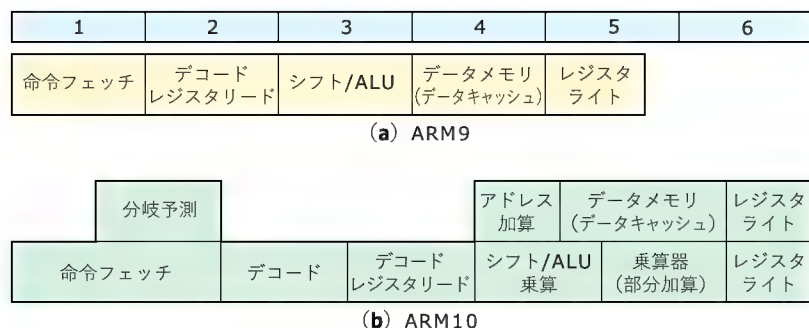
1999年に発表されたSH-5であるが，2002年になって初めて展覧会などでプロトタイプのデモが行われるようになった．動作周波数は400MHzにはほど遠い256MHzである．SH-6やSH-7についても計画が発表されているが，ここでは省略する．

3 ARM/StrongARM/XScale

● ARM7までとARM8

最初のARMアーキテクチャのMPUが開発された当時，RISCはスタンフォード大学のMIPSと，カリフォルニア大学バークレー校のRISC I，II（SPARCの母体）しか例がなかった．ARMはバークレーRISCを参考にして設計された．ロード/ストアアーキテクチャ，32ビット固定長命令，3オペランドフォーマットという特徴を採り入れたが，レジスタウィンドウ，遅延分岐，全命令の1クロック実行は採用しなかった（ほとんどの命令は1クロックで実行するが）．設計目標は，CISCライクな命令セットをRISCに準じた単純なハードウェアで実行することに置いている．命令セットの特徴はソースオペランドをシフトした後に演算可能なこと，ほとんどすべての命令が条件コードを変更し，条件コードに応じた処理が可能なことである．

ARMにはARM1～7，ARM8，StrongARMとアーキテクチャに若干の差異がある．ARM1～7は単純な3ステージのパイプラインを基本としていたが，その後改良が重ねられ，ARM8で標準的な5ステージのパイプラインにたどり着く．

〔図4〕ARM系プロセッサのパイプライン

〔図5〕ARM9/ARM10のパイプライン

ARM8では，パイプラインへの命令供給のバンド幅を向上させるため，命令のプリフェッチを行いバッファリングする．初代のARM8のプリフェッチユニットには静的な分岐予測機能も内蔵されていたという．**図4(b)**にARM8のブロック構成を示す．パイプラインは次の5ステージから構成される．

1）命令プリフェッチ
2）命令デコード，レジスタリード
3）実行(シフトと演算)
4）メモリアクセス
5）ライトバック

● ARM9/ARM10

ARM8の後継であるARM9のパイプラインは，ARM8とほとんど同じである．その後継のARM10ではパイプラインに変更が加えられた(**図5**)．つまり，高い動作周波数を実現するために，デコード部と実行部のステージを2段に分割している．キャッシュアクセスは1.5段分をかけてアクセス時間に余裕をもたせている．また，アドレス計算用の加算器を専用にもち，ロード/ストアのパイプラインを整数演算系と分離している．これにより，データキャッシュはノンブロッキング(ヒットアンダミス)が可能になっている．さらにARM10では，パイプラインのステージ数増加による性能低下(CPIの増加)を低減するため，動的な分岐予測が採用された．ARM社は，これによりARM10はARM9と同等なCPIが得られるとしている．

● StrongARM

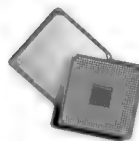
ARMのパイプラインはARM社とDEC社が共同開発したStrongARM(現在はIntel社に買収されている)で一応の完成を見る．キャッシュの構成が命令とデータに分割された(命令とオペランドフェッチで待ち合わせが生じない)こととレジスタのフォワーディング機能が追加されたのが特筆すべき特徴である．パイプラインは次の5ステージで構成される．

1）命令フェッチ(命令キャッシュから)
2）命令デコードとレジスタリード，分岐先のアドレス計算
3）オペランドのアドレス計算，またはシフトおよび演算を実行
4）データキャッシュへのアクセス
5）レジスタファイルへ結果をライトバック

図4(c)にStrongARMのパイプライン構成図を示す．

ARMのパイプラインも命令ごとに可変なステージ数から始まり，結果として5ステージに落ちついたようである．やはり，5ステージというのがRISCのパイプラインの王道といえるのかもしれない(少なくともこれまでは)．

● XScale

Intelから発表されたXScale(かつてはStrongARM2と呼ば

れた)では，600MHz(当初の目標)という高い動作周波数を実現するため，再びパイプラインの見直しがなされた．結果，整数演算で7ステージ，ロード/ストアで8ステージという構成になった〔**図4(d)**〕．ステージ数はそれほど多くはないが，インテルはこれをスーパーパイプラインと呼んでいる．

パイプラインが2ステージ増えた理由は，おもに2本のクリティカルパス(タイミングネックになる論理経路)対策のためである．一つ目はALU演算である．従来のStrongARMでは1クロックで

シフト→ALU演算→条件コードの生成

を行っていた．これを3ステージに分割して処理する．こうすることにより，命令デコードにも余裕ができた．従来は命令デコードとレジスタアクセスを1クロックで行っていたが，

レジスタアクセス→シフト

のタイミングを，従来より，遅らせて余裕をもたせている．

二つ目はデータキャッシュのアクセスである．データキャッシュは，従来は，

アドレスデコード→キャッシュアクセス→データの整列→ALUへ入力

を1クロックで行っていた．XScaleではデータキャッシュが従来の2倍の32Kバイトになったので，一度に動作する回路が多くなりクリティカルパスになった．そこでデータキャッシュアクセスを2クロック(2ステージ)で行うように改良した．

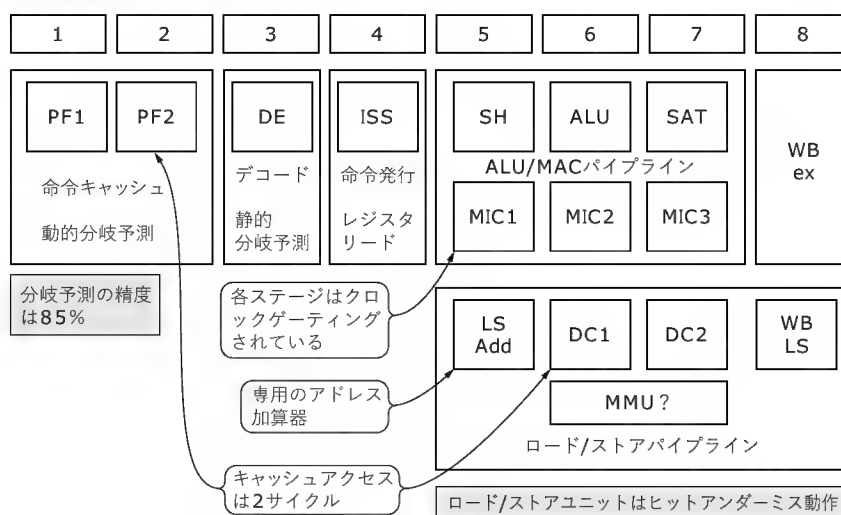
XScaleではパイプラインのステージが増加したため，分岐命令の性能低下(当然，分岐予測機構は備えている)などを考慮するとCPIが5～8％増加するが，周波数を1.5倍に向上することで，差し引き40％程度の性能向上となる．なお，分岐ターゲットバッファは128エントリからなるダイレクトマップキャッシュで，2ビットの情報で分岐の履歴を管理する．

〔**図6**〕**ARM11のパイプライン**

● ARM11

一方，ARM社は2002年4月にARM11の概要を発表した．8ステージのシングルパイプラインで350～500MHz動作を目指す．明らかにXScale対抗が見てとれる．**図6**にARM11のパイプラインを示す．ARMアーキテクチャのクリティカルパスは，XScaleでも説明したが，シフト＋ALUの同時実行，キャッシュアクセス，そしてMMUにある．これらのステージを独立化することで，高速動作を実現できる．基本的な考え方はXScaleのパイプラインとよく似ている．

4 V800シリーズ

V800シリーズはNECがV80の後継として開発した，どちらかといえば，マイクロコントローラといえるMPUシリーズである．その開発目標は，低価格で低消費電力のチップであった．V800シリーズでは基本の命令長を16ビットとしながらも，大きいビット数のイミディエート値やディスプレースメントの指定でコード効率を上げるために，32ビットの命令長も用意している．ちょうどSHとARMの中間のようなアーキテクチャである．

V800シリーズは，1992年に最初のV810が開発され，その後V830，V850が続いて開発された．現在は，これらのMPUをCPUコアとした周辺内蔵品が販売されている．

V810のパイプラインに関しては，ユーザーズマニュアルには記載されていない．しかし，当時の雑誌の解説記事によると，

フェッチ，デコード，実行，書き込み

という典型的(と記述されている)なRISCのパイプラインに対し，デコードと実行の間にレジスタリードのステージを挿入した，

1) フェッチ
2) デコード
3) レジスタリード
4) 実行
5) 書き込み

という構成だという．これは，可変長の命令フォーマットをデコードするのに長い処理時間が必要であり，デコードに余裕をもたせるためと説明されている．合計5ステージ構成のパイプラインであるが，これは33MHz以上の動作周波数を想定したものであり，25MHzの動作周波数ではレジスタリードのステージは結果として不要だったようだ．

しかし，このパイプライン構成では，ロード/ストア命令でのオペランドフェッチステージがない．おそらく，ロード/ストア命令の処理時には6ステージになるのであろう．

その後，V830/V850になるとパイプラインの見直しが行われ，

1）IF：命令フェッチ
2）RF：命令デコード
3）EX：実行
4）MEM：オペランドアクセス
5）WB：ライトバック

という5ステージのパイプラインになった．これは何度も説明している典型的なRISCのパイプラインと同じで，とくに取り立てていうこともない．ただ1点，分岐命令は，TAKENする場合，EXステージの終了を待ってIFステージを開始する．これは前章の図9(c)と同じである．明らかに分岐先のアドレス計算から命令フェッチまでに時間的余裕をもたせているのがわかる．このため，分岐命令のレイテンシは3クロック（NO TAKENの場合は1クロック）になる．

V800シリーズでは遅延分岐を採用しないため，分岐が多いプログラムの処理は不利になる．また，歴史的には古いMPUのせいか，分岐予測機構も採用していない．シンプルイズベストという考え方なのだろう．

5 R4000

R4000はスーパーパイプライン構造を採用し，高い動作周波数で動作させることを目的としている．パイプラインはIF，IS，RF，EX，DF，DS，TC，WBの8ステージで構成され，（筆者予想では）単相クロックに同期して動作する．**図7**にR4000のパイプラインの詳細を示す．各ステージでの動作は次のようになっている．

1）IF

命令フェッチ1段目．命令の仮想アドレスが命令キャッシュとTLBに転送される．

2）IS

命令フェッチ2段目．命令キャッシュが命令を出力し，同時にTLBは命令の物理アドレスを出力する．

3）RF

レジスタファイル．次の3動作が並行に行われる．

a）命令をデコードし，インタロック条件をチェックする
b）命令キャッシュのヒット/ミスがチェックされる
c）レジスタファイルからオペランドをフェッチする

4）EX

命令実行．次の3動作の一つが実行される．

a）命令がレジスタ-レジスタ間命令なら演算を実行する
b）命令がロード/ストア命令ならオペランドの仮想アドレスを計算する
c）命令が分岐命令なら，分岐先の仮想アドレスを計算する．同時に分岐のTAKEN/NOT TAKENを決定する

5）DF

データキャッシュ1段目．オペランドの仮想アドレスがデータキャッシュとTLBに転送される．

6）DS

データキャッシュ2段目．データキャッシュが値を出力する．同時にTLBはオペランドの物理アドレスを出力する．

7）TC

タグチェック．ロード/ストア命令の場合，データキャッシュのヒット/ミスをチェックする．

8）WB

ライトバック．命令の実行結果をレジスタファイルに書き込む．ストア命令の場合はデータキャッシュに書き込む．

R4000の各パイプラインステージは基本的には1クロックであるが，時間がかかるキャッシュアクセスには時間をかけている（タグチェックを含めて3クロック）．R4000の発表当時はスーパーパイプラインとしてクローズアップされたが，現在においてはごく普通のパイプライン構成である．

R4000ではパイプラインが8ステージになったため，分岐命令の実行時に3クロック，ロード命令の実行時に2クロックの遅延スロットが生じる．分岐命令においてはR3000と互換性をもたせるため遅延スロットの1命令分は実行するが，残りの2クロックはバブル（むだな時間）になる．分岐命令の実行時間がR3000の1クロックから3クロックになった（遅延スロットを含まない）と思えばよい．

ロード命令においては遅延スロットに相当する後続2命令がロード命令のデスティネーションオペランドと一致している場合はインタロックが生じる．つまり，R4000では遅延ロードを採用しない．さすがに，ロード命令とその結果を使用する命令の間を2命令分も空けるのは現実的でないと考えたのであろう．

分岐命令の実行時間を短縮するため，R4000ではLikely分岐（Branch Likely）が導入された．Likely分岐とは，分岐条件が成立するときのみ遅延スロットの命令を実行する条件分岐命令である．分岐条件が成立しなければ遅延スロットは無効化される．遅延スロットにNOP命令があると考えてもよい．分岐

〔図7〕R4000のパイプライン

	IF	IS	RF	EX	DF	DS	TC	WB
	ICache		ID	ALU	DCache		TC	WB
IVA	ITLB		RF	DVA	DTLB			
			TC					

ICache Access　　DCache Access

IF：ICache First	IVA：Instruction Virtual Address
IS：ICache Second	ICache：ICache Access
RF：Register File	ITLB：Instruction TLB
EX：Execute	ID：Instruction Decode
DF：DCache First	ALU：ALU Operation
DS：DCache Second	DVA：Data Virtual Address
TC：Cache Tag Check	DCache：DCache Access
WB：Writeback	DTLB：Data TLB

〔図8〕Topaz(24K)のパイプライン

命令がループ処理の終わりにあるような場合，分岐命令をLikelyにして分岐先の1命令を遅延スロットに置けば，ループ内の命令が1命令減少するので，実質的に分岐命令の実行時間を短縮できる．これは一種の(静的な)分岐予測とみなすこともできる．

Likely分岐はR4000以降のMIPSアーキテクチャで採用されているが，スーパースカラ構造では実装が難しいせいか，将来的には削除したい意向だという．その前兆か，MIPS-3Dという拡張アーキテクチャで採用された，bc1any2，bc1any4という条件分岐命令ではLikely分岐が定義されていない(命令コード的には割り当て可能だが)．

6 Topaz(24K)のパイプライン

それまでMIPS社はIPコアビジネスに注力してきたが，2003年6月に方向転換を行った．従来，32ビットアーキテクチャのMIPS32と64ビットアーキテクチャのMIPS64の2系統のアーキテクチャを管理してきたが，64ビットアーキテクチャの需要がないのか，32ビットに特化するようになった．それまで，MIPS64系のIPコアとしては，5K(Opal)，20K(Ruby)，25K(Amethyst)が存在したが，最新のロードマップには5Kのみが掲載されている．従来，20Kや25Kがあった位置には，新たに24K(Topaz)が掲載されている．24Kという名称からも，性能的に，25Kを置き換えるという意味合いが感じられる．

24Kは，6月17日に発表したMIPS32アーキテクチャの論理合成可能なIPコアである．8ステージのシングルパイプラインで動作周波数目標は400～500MHzである．最初からマルチプロセッサに対応し(キャッシュはMESIアルゴリズム)，マルチコアで性能向上を目指す．2004年からライセンスを開始する．

MIPS社から発表された24Kのパイプラインを**図8**に示す．パイプライン自体には不明な点が多いが，とりあえず発表された情報を説明する．

1) IF

命令キャッシュ第1段階．

2) IS

命令キャッシュ第2段階．最大64Kバイトの4ウェイキャッシュに対応(16Kバイト/1ウェイ)し，命令キャッシュは2サイクルのレイテンシ(RAMアクセスに1サイクル，タグチェックとウェイ選択に1サイクル)でアクセスする．命令キャッシュからは2命令を同時にフェッチ可能で，結果は6エントリの命令キューに格納される．これにより，命令フェッチと実行がデカップル構成になり，高い命令発行レートを維持できる．最大2命令までのノンブロックリードが可能であり，命令キャッシュへのリフィル時間を最適化できる．動的な分岐予測(512エントリの分岐履歴テーブル)，4エントリのリターンスタックを備える．分岐予測ミス時のペナルティは4クロックである．

3) RF

レジスタファイルへのアクセス．

4) AG

アドレス生成および命令発行を行う．整数演算とメモリパイプラインを分離することで，データキャッシュは4エントリのノンブロックリードに対応する．なぜ整数パイプラインにアドレス生成が必要かは，不明(分岐アドレスの生成用か)．

5) EX

命令実行．

6) MS

乗算命令とシフト命令処理用の追加ステージ．32ビット×32ビットの乗算が，1クロックのリピートレート，5クロックのレイテンシで実行できる．

7) ER

結果の整列(ローテート)と符号/ゼロ拡張を行うステージ．

8) DCache

データキャッシュへのアクセス．クリティカルワードをフォワーディング可能(キャッシュバイパス)．

9) Sel

データキャッシュのタグチェックとウェイ選択．

10) WB

結果をレジスタへライトバック．

24Kのパイプラインの特徴としては，将来の高速動作を見越して，命令フェッチとデコードをデカップル構成にしたことであろう．シングルパイプラインでは珍しい．

なかもり・あきら　フリーライタ

Appendix 1

システムオンチップ時代のデバッグ手法

エミュレーション機能の基礎

中森　章

はじめに

ほとんどのMPUがもっていてマニュアルには通常記載されていない機能に，エミュレーション機能がある(デバッグ機能ともいう)．

ほかのアーキテクチャの命令コードを実行することもエミュレーションというが，ここではデバッガであるICE(In-Circuit Emulator)を実現する機能のことを指すこととする．本稿では，エミュレーション機能の概略を説明する．

● ICEとは

ICEとは，一言でいうとデバッガである．しかし，いわゆるソフトウェアのみで実現されているGDBのようなデバッガと違い，リアルタイム(実時間)エミュレーションが可能である．これは，実チップをターゲットシステムに実装してエミュレーションを行うことにより，実デバイスと同じAC特性やDC特性を実現したままのデバッグを可能にするものである．つまり，実チップと同じタイミングや環境でデバッグができる．

ICEは，実チップの代わりに専用プローブを実際のボードに差し込むことで，実チップの動作をエミュレートする(**図A**)．ユーザーからは，ICEというシステムが一つの実チップに見える．

その昔(今でも？)，ICEは高価だったので，それを代替する手段もいくつか考えられている．たとえば，ロジックアナライザ(ロジアナ)をエミュレータの代わりにデバッガとして使う手法である．ロジアナで取り込んだバスサイクルを逆アセンブルして，実行する命令列を表示するソフトウェアはけっこう活用されていた．しかし，この手法はICEにはかなわない．

● エミュレーション機能とは

エミュレーション機能とは，ICEを実現するためにMPUが提供する機能のことである．デバッグ機能ともいう．具体的には，アドレスやデータの値によるトラップ機能(ハードウェアブレーク)や命令実行のトレース機能を指す．MMUを内蔵するMPUでは仮想アドレスを出力する機能もある．ハードウェアブレークとは，ブレークポイント命令をプログラムに埋め込んで，そこを通過した場合にブレークする，ソフトウェアブレークとは対称的に，ハードウェア自身が備えるブレーク機能のことである．

〔図A〕ICEの構成

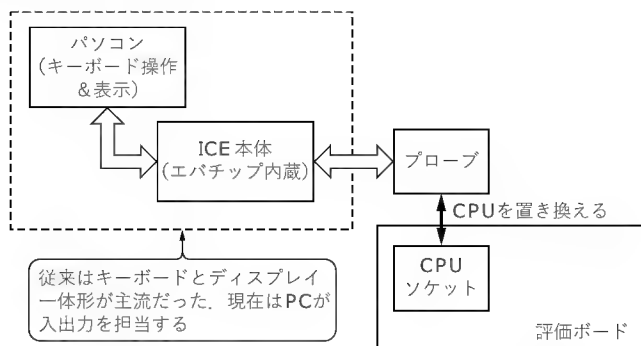
エミュレーション機能を実現するためには専用端子が必要なので，通常のチップ(通称本チップ)よりも端子数を増やしたエバリュエーションチップ(通称エバチップ)が製造される場合もあるが，基本的には本チップとエバチップは同一のダイ(チップ)であることが多い．つまり，外部端子に接続されてないだけで，ユーザーが手にする本チップもエミュレーション機能を内蔵している．しかし，その機能の使い方を知る方法はない．エバチップは，ICEメーカーに対して出荷される専用チップである．

最近の流行は，エバチップを作らず，本チップにエミュレーション機能を内蔵させることである．この場合は，ユーザーがその機能を使おうと思えば使うことも可能である．ただし，エミュレーション機能の詳細はICEメーカー以外には公開されないのが普通なので，現実にユーザーが利用するのは難しい．

● フォアグラウンドモニタとバックグラウンドモニタ

GDBなどの通常のソフトウェアデバッガとICEとが決定的に異なるのは，制御プログラムがユーザーの資源を占有するか否かである．ソフトウェアデバッガはユーザーのメモリ空間にロードされ，一つのタスクとして目的のプログラムをデバッグする．しかしICEは，通常は，制御プログラムのためにユーザーのメモリ空間を必要としない．これは，より現実に近い環境でプログラムをデバッグできるという利点のほかに，プラットホームの立ち上げ時，ボードにROMやRAMがまだ実装されてない段階でも，プログラムを実行することができるという利点がある．

ICEは二度美味しい．ボード設計の段階ではハードウェア屋が，ROMやRAM，あるいは外部I/Oなどに正常にアクセスできるかという，ボードのハードウェアのデバッグに使用できる．ROMやRAM上にプログラムが存在する必要はない．ボードが完成したあとは，ソフトウェア屋がソフトウェアをデバッグするのに使用できる．ソフトウェアのデバッグにおいては，ソフトウェアデバッガで十分という意見もあるかもしれない．しかし，ICEを使えば，ブレークポイント命令を埋め込むことのできないROM領域でブレークさせることもできるし，ある特定のアドレスに対してロードやストアを行った場合にブレークさせることもできる．あるいは，実時間で命令実行のトレースを行える．

さて，ICEの制御プログラム(モニタプログラム)はユーザー空間とは別の空間に置かれる．これは，ハードウェアブレーク発生時に，専用端子を活性化することで，ハードウェア的にアドレス空間の切り替えを行うことで実現される〔**図B(a)**〕．あるいは，モニタプログラムを実行するための特殊な空間がMPUに内蔵されている場合もある．このような方式をバックグラウンドモニタと呼ぶ．簡易的なICEでは，モニタプログラムをユーザー空間に置く場合もある〔**図B(b)**〕．

このような方式をフォアグラウンドモニタと呼ぶ.

● キャッシュ非内蔵時のエミュレーション機能

MPUの提供するエミュレーション機能は，MPUがキャッシュを内蔵しない場合とする場合で若干異なる．まずは，キャッシュを内蔵しないMPUが提供する，一世代前のエミュレーション機能について解説する．

(1) アドレストラップとデータトラップ

これは，MPUがアクセスするアドレスを指定してトラップを発生させる機能である．トラップが発生した後，制御はICEのモニタプログラムに移る．

キャッシュを内蔵しない場合，すべてのアクセスは外部バスに出力される．このため，バスサイクルを監視する機構を設けておけばアドレストラップを実現するのは難しくない．目的のアドレスやデータが出力されたら，ICE空間に移行させるための強制ブレーク機能をMPUが備えていればよい．

より細かい条件でトラップを発生させるためには，現在起動されているバスサイクルの種類を表示するステータス出力があればよい．

(2) ステップ実行

1命令ずつ実行しながらレジスタ内容やステータスを表示する機能である．この機能は，アドレストラップを次の命令のアドレスに設定することで実現する．

(3) 仮想アドレス出力

本チップには端子数の制限から，仮想アドレスを出力する機能はない．しかしエバチップには，物理アドレスと同時に仮想アドレスを出力する機能がある．

専用のエバチップなしで仮想アドレスを出力するための手法としては，バスサイクルごとに仮想アドレスと物理アドレスを時分割で出力することが考えられる．しかし，本チップでは仮想アドレスの出力を行わないため，本チップとエバチップ(もしくはエミュレーションモード)でタイミングに差異が出るという欠点がある．

(4) トレース機能

トレース機能は，プログラムのデバッグには非常に有用な機能であるが，実現は難しくない．数十MHz以上で変化するMPUの状態をリアルタイムでユーザーが認識することは不可能なので，トレース機能はある程度までを実行した後に，そこまでの実行履歴を調べる場合に使用する．したがって，バスサイクルの出力をそのまま保存しておくことができれば，あとはソフトウェアでどうにでもなる．つまり，

- バスをトレースし外部バッファに蓄える
- そのデータを逆アセンブルなどの加工を施して表示する

という手順で行うことができる．どの程度の命令数をトレースできるかは外部バッファの容量による．

(5) 分岐トレーストラップ

プログラムをデバッグするときに有用なのは，サブルーチンなど，命令の流れが不規則に変化する時点を認識することである．分岐の履歴を調べれば，プログラムの大まかな流れを知ることができる．

この機能を実現するために，分岐トレーストラップを提供するMPUもある．これは，分岐が発生した時点でトラップを発生させる機能である．ただ，分岐でいちいちトラップを発生させていると，プログラムの動作が実時間で動作させた場合と異なるのが欠点である．

(6) シーケンシャルブレーク

シーケンシャルブレークとは，その名のとおり，あるトラップ条件が成立した後に別のトラップ条件が成立するとき，初めてトラップさせる機能である．この機能は，トラップ発生時にICEの実行を止めるか否か，ICEのモニタプログラムで制御することで実現可能である．つまり，1回目のトラップが発生すると，そのことを記録しておき，そのままユーザープログラムに制御を戻す．そして，再びトラップが発生したときに，その旨をユーザーに表示すればよい．

〔図B〕バックグラウンドモニタ/フォアグラウンドモニタ

(a) バックグラウンドモニタ

(b) フォアグラウンドモニタ

ソフトウェア制御できるため，一見MPUがもつまでもない機能と思えるが，プログラムの動作を実時間で動作させた場合と一致させるためには必要な機能である．

● キャッシュ内蔵時のエミュレーション機能

キャッシュの内蔵は，MPUの性能に飛躍的な向上をもたらした．しかし，ICEにとっては嬉しいことではなかった．なぜなら，従来ICEが拠り所にしていたバスサイクルが発生しなくなったからである．たまに発生するバスサイクルもほとんどがキャッシュリフィルのためのバスサイクルで，バスサイクルとそれを発生させた命令を1対1に関連付けるのは難しい．このため，キャッシュ内蔵を当たり前とするRISCプロセッサ用のICEは，いつまで経っても実用的なものが登場しなかった．

当初RISCは，ワークステーションなどハイエンドの分野でしか使用されていなかった．この分野のデバッグは，昔ながらのロジアナで波形観測を行う手法が普通であり，ICEの必要性を訴える人は少なかった．まさに職人芸の世界である．しかし，RISCが組み込み制御にも使われるようになるとICEの要望が高まってきた．組み込み分野では，従来ICEを使用してデバッグを行っていたので，「ICEのないMPUなんて使えない」という意見が多数派だったのだ．

このような事情もあり，キャッシュ内蔵のMPUでは，エミュレーション機能を実現するために新たな機能を内蔵することが必要になった．たとえば，MotorolaがDragonBallやColdFireシリーズに内蔵したBDM(Background Debug Mode)がそれである．BDMはCPUのマイクロコードでデバッグ命令を実装し，専用のデバッグ端子を外部にもたせて，専用のケーブルでデバッガと交信するしくみになっている．

(1) ハードウェアブレーク

従来，ICEがバスサイクルを観測することで実現していた，アドレストラップ，データトラップという機能をMPUに内蔵するように

なった．従来も，アドレストラップ機能を内蔵し，ROM領域でもブレークポイントを設定できることを売りにするMPUはあったが，ロードやストア時のデータの値を指定してトラップさせる機能をもつものはほとんどなかった．

MPUは，ユーザーが使用するアドレストラップのほかに，ICE専用のアドレストラップやデータトラップ機能を提供するようになった．

トラップ発生時のICE専用空間への移行も，MPUがサポートする．

(2) トレース機能の実現

トレース機能も，従来はICEが外付けで実現していた機能をMPU内に取り込むことで理論上は可能である．しかし，それはMPU内部に巨大なトレースバッファを内蔵することを意味する．

ICE以外では使用しない機能でチップの面積を増大させるのは，好ましいことではない．そこで，最小限のハードウェアでトレース機能を実現する方式がいろいろ考案された．その基本原理は，分岐が発生したことを検出する点にある．従来一部のエバチップで採用されてきた分岐トレーストラップ機能はとくに有用である．

トレーストラップ機能に加え，分岐時に分岐先の仮想アドレスを出力できれば，ほぼ完全に命令実行をトレースできる．しかし，この仮想アドレスの出力というのが難問である．従来のエバチップでは余分な端子数を追加することで実現していた機能である．本チップに，たとえば32本の仮想アドレス端子を追加することは不経済である．そこで考案されたのが，4本程度の専用端子を追加し，時分割で仮想アドレスを出力するという方式である．32ビットなら8回に分けて出力する．この方式の欠点は，分岐が連続して発生すると，仮想アドレスの出力が命令実行に追いつかなくなり，情報が失われることである．しかし，RISCの初期のICEでは，ないよりはまし，という割り切りで採用されていた．

実際，トレース機能の実装はいろいろな制限から難しいことが多く，ハードウェアブレークだけでICEが作られることも多い．バックグラウンドモニタ機能を提供すれば，ハードウェアブレークしかできなくても，そこそこ使えるICEが構成できる．

● JTAGを利用したデバッガ

JTAGは，国際標準規格IEEE1149.1として普及している．JTAGは，機能の名称ではなく，この規格化作業を推進したグループの名称である．機能の名称は「バウンダリスキャン」である．IEEEの標準では，バウンダリスキャンアーキテクチャとそれにアクセスするためのシリアルポート（通称JTAGポート）が規格化されている．このJTAGポートは，本来，ボードのテスト用に考案されたものであるが，そのインターフェースをMPUのデバッグ機能に利用することが考えられている．

従来のICEでは，専用のデバッグ端子をMPUに装備する手法が採られていたが，現在はJTAGを利用する方法が主流である．JTAGは，もともとチップの回路テスト用に開発されたバウンダリスキャンのテスト手法である．JTAGの回路と外部テスト端子を利用し，デバッグ回路を追加することによりシステムのデバッグを行う．

JTAGを利用したデバッグ機能の拡張は，各社独自の仕様により行われている．こうしたオンチップデバッグ機能には，NECのN-WIRE，MIPS Technologies社のEJTAG（Enhanced JTAG），Motorola社のCOP（Common On Chip Processor）などがある．本質はどれも似たようなものである．本来，N-WIREというのは，ICEメーカーであるHPが提唱したJTAGを使ったデバッグ機能であり，その意味ではNEC固有の規格ではない[注]．

ICEの短所は，ソフトウェアデバッガに比べて非常に高価な点である．従来は，100万円を超えることは珍しくなかったが，JTAGを利用したデバッグ機能は，MPU内にデバッグ機能を内蔵し，外部とのインターフェースを最小限の端子本数（5〜15本程度）におさえることで安価（30〜50万円）なICE構築をめざすものである．

JATGと同時に用いられるオンチップのデバッグ機能は，従来と大差ないハードウェアブレークとトレース補助機能である．それを，JATGという標準的なシリアルインターフェースでアクセスすることで，ICEのハードウェアの共通化を図ることができる．つまり，JTAGの先のデバッガ本体は，MPUの種類が変わっても同一のものを用いることができる（**図C**）．

JTAGのICEでは，従来のICEとは異なり，MPU自体がデバッグ機能を内蔵するので，エバチップは不要である．ボードテスト用のJTAG端子をデバッガに接続することでデバッグが可能になる．

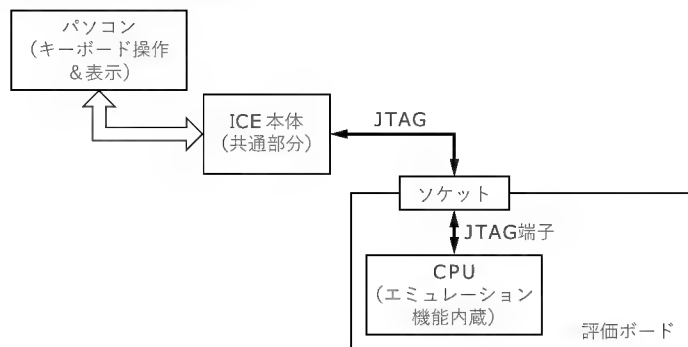
● JTAGデバッガの実現例

JTAGデバッグ機能を有するMPUの一般的な構造を**図D**に示す．このように，MPUのデバッグユニットはバックグラウンドモニタ（ハードウェアブレーク機能とモニタ機能）とトレース制御から構成される．

MPUが提供するハードウェアブレーク機能は直感的に理解しやすいが，トレース機能が実現される方法は興味深い．ここでは，JTAG ICE（N-WIRE）におけるトレース機能とモニタ機能の実現方法を簡単に説明する．

(1) トレースパケット

命令やデータのトレースは，トレースパケットと呼ばれる数種類の情報を，JTAGを通じて数ビットずつシリアル出力することで実現する．トレースパケットはMPU内部で生成される．トレースパケットの例を**図E**に示す（実際のものとは異なる）．これは，パケットの種類を判別するTRCODEとそれに付随する情報で構成される．これ

〔図C〕JTAG ICEの構成

注：N-WIREは東芝のMIPS RISCであるR3900に初めて搭載された．東芝とYHP（横河ヒューレットパッカード）が開発した規格で，*N*本の信号線から構成されるので，そう命名された．その後，HPがJTAG仕様に変更しさらなる拡張を行って現在の形になった．このとき，ハードウェアブレーク部とトレース部（N-Trace）が分離された．つまり，トレース機能のないものもN-WIREと呼ぶ．NECやARMも採用しているが，ARMはN-Traceと呼んでいる．また，日立のSH-3/SH-4に採用されたH-UDI（Hitachi User Debug Interface）も同様の形式であるらしい．

らのトレースパケットは，アドレスの一致情報，分岐や例外の発生情報，その分岐先アドレスや分岐元アドレス，例外コードを示す．必然的に分岐に関する情報が多い．あとは，外付けのICEがトレースパケットを取り込んで，従来どおりのトレース表示を行う．

ICEは通常，低いクロックで動作している．反面，MPUは非常に高速なクロックで動作する．トレース機能は高速なMPUの状態をパケットにして出力するものであるから，ICEがそれを取り出して処理するためには，速度差を緩衝するバッファが必要である．まともなトレース機能を実現するためには，このトレースバッファのために多くの容量を必要とし，これがMPU内にトレース機能を実装する足枷となることが多い．このため，トレースバッファをMPUの外部メモリとしてサポートする場合（MIPSのEJTAGなど）もある．この場合はMPUの動作クロックをあまり高速にできないのが欠点である．

（2）モニタ機能

デバッグ機能を有するMPUはモニタ空間を内蔵する．つまり，デバッグ機能を実現するプログラム（モニタプログラム）はモニタ空間で実行される．しかし，4Kバイトや8Kバイト程度のモニタプログラムのためのデバッグ専用メモリを内蔵するのは不経済である．そこで考案されたのが，一つのモニタ機能を1～8命令程度で実行するものとし，その命令分の実行領域のみを内蔵する方式である．この領域をモニタ命令レジスタと呼ぶ．このレジスタは1回に連続実行する命令の数だけ存在する（たとえば8本）．そして，具体的には，次のような手順でモニタ命令レジスタの内容を実行する．

1）アドレストラップやハードウェアブレークなどでモニタ空間に移行する
2）命令実行が自動的に停止する
3）JTAGを経由して，モニタ機能を実現する1～8命令程度の命令列を，モニタ命令レジスタに書き込む
4）JTAG経由で命令実行を許可する
5）モニタ命令レジスタの内容が実行され，実行が終わると命令実行が停止する
6）モニタ命令レジスタの実行結果は一時的なレジスタに格納され，その値をJTAG経由で取り出し，実行結果などの表示を行う
7）3）～6）の処理を繰り返す
8）モニタ空間から抜け出すための命令をモニタ命令レジスタに書き込む
9）JTAG経由で命令実行を許可する
10）制御はユーザープログラムに移る

〔図D〕JTAG ICEの動作

〔図E〕トレースパケットの例

まとめ

MPUの提供するデバッグ機能とICEの実現方法の概要について説明してきた．最近ではICEの提供が，MPUを販売する上での必須事項になりつつある．これを実現するために，MPUにどのような機能が要求されるのかを知っておくことは，有用であろう．

なかもり・あきら　フリーライタ

Chapter 4

1クロックで複数の命令を同時に実行する

並列処理の基本とスーパースカラ

中森　章

ここでは，シングルパイプラインを多重化したスーパースカラについて解説する．1命令1クロック処理があたりまえになってくると，プロセッサの性能はクロック数（と命令の機能）だけで決まってしまい，アーキテクチャ的には進化の余地はないように思える．そこで登場するのが，1クロックで複数の命令を同時に実行してしまおうというアプローチだ．その代表がスーパースカラという手法であり，現在の高性能MPUの多くで採用されている．ここでは，スーパースカラの基本的な考え方をおさえておこう．

（筆者）

1 CPIからIPCへ

CPI（Clock cycles Per Instruction）とは，1命令を実行するのに必要なクロック数である．この値が小さいほどMPUは高性能であるといえる．CISCからRISCへの進化によって，CPIが1という限界に達した（理想的な実行環境に限定されるが）．それ以上性能を上げるには，同じパイプラインステージ内で複数の命令を実行させればよいと考えるのが自然な発想である．これがスーパースカラである．そうなってくると，性能指標としてCPIの逆数である**IPC**（Instructions Per Clock cycle）を使用したほうがわかりやすい．つまり，1クロックに実行できる命令数である．2命令を並列に実行できればIPCは2に近づくし，4命令を並列に実行できればIPCは4に近づく（理論的には）．

IPCは値が大きいほど高性能を表す．IPCは**MIPS**（Million Instructions Per Second）値とも密接な関係がある．MIPS値とは，1秒間に実行できる命令数（100万命令単位）である．その意味で，IPCに動作周波数（MHz単位）を掛け算した値がMIPS値である．つまり，IPCが1の場合，100MHz動作では100MIPS，200MHz動作なら200MIPSである．

もっとも，最近のMPUのMIPS値は**Dhrystone MIPS**を採用しているので，公称性能は本来の意味のMIPS値とは異なる．Dhrystone MIPSとは，有名なミニコン（死語）であるVAX-11/780の性能を1MIPSとし，Dhrystoneベンチマークを実行したときの性能がその何倍の値になるかを示したものである．Dhrystone MIPSを用いれば，シングルパイプラインでもIPCが1を超えているように見えるので多用される．

しかし，スーパースカラ構造になると事情が異なる場合もある．Dhrystone MIPSを用いると性能がそれほど高く見えないからだ．その代わり，IPCと同時実行できる命令の数が等しいと仮定して，たとえば，2命令同時実行可能なパイプラインを200MHzで動作させると400MIPS（200MHz × 2命令という計算）などという理想値を示す場合もある．現実には，同時実行できる命令数を増やしていっても，IPCは1.6あたりに収束することが経験的にわかっているので，2命令同時実行でもIPCが2になることはまずない．

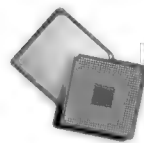
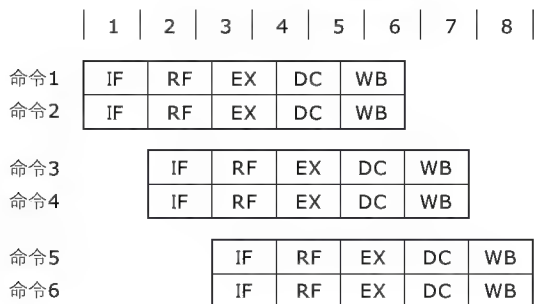
ただし，Dhrystone MIPSを真のMIPS値と（意図的に）混同してIPCを計算すれば，2命令同時実行で2.2程度になることもある．この場合でも，4命令同時実行では4.0どころか3.0を越えることはまずない．その場合は，動作周波数 × 4でMIPS値が決められたりする．つまり，200MHzで動作し，4命令同時実行なら800MIPSといった具合である．まあ，公称MIPS値をそのまま信じる人はいないと思うが，このような数字のマジックに惑わされないようにしなければならない．

しかし，感覚的にはIPCが2.2などといわれると非常に高性能と思ってしまう．現在のGHz単位で動作するx86系のMPUのIPCは2～3などといわれているが，Dhrystone MIPSによるIPCでは0.6程度である（つまり実質的な性能は，動作周波数の割には高くない）．

一般にパイプラインのステージ数を増やすと，IPCは低下する．動作周波数を向上するためにパイプラインのステージ数を増やすことはよくある手法だが，パイプラインのステージ数を増やしてもIPCを0.6程度に保ち続けているインテルやAMDは賞賛に値する．NetNews（`fj.comp.arch`）に，PentiumIII-750MHzでDhrystoneベンチマークを行った場合の性能の実測値が報告されていた（メッセージIDは失念した）．その値から計算すると，真のIPCは1.01，Dhrystone MIPSによるIPCが1.18であった．予想の2倍の性能になっているが，これはDhrystoneという，最高性能を発揮しやすいプログラムの性質によるものだろう．実際のアプリケーションではこうはいくまい．

ちなみに，別の資料によるPentium（P5）66MHzのDhrystoneによるIPCは1.5なので，PentiumIIIになるとIPCは低下している．パイプラインのステージ数が増加しているので，当然といえば当然か．

〔図1〕スーパースカラ(2ウェイ)の概念図

	1	2	3	4	5	6	7	8
命令1	IF	RF	EX	DC	WB			
命令2	IF	RF	EX	DC	WB			
命令3		IF	RF	EX	DC	WB		
命令4		IF	RF	EX	DC	WB		
命令5			IF	RF	EX	DC	WB	
命令6			IF	RF	EX	DC	WB	

とにかく，シングルパイプラインの目標がCPIを1に近づけることであったように，スーパースカラの目標はIPCを同時実行できる命令数に近づけることである．まあ，x86は独自の道を歩んでいるようにも思えるが．

2 複数の命令を並列実行するスーパースカラの概念

複数の命令を並列実行する機構を**スーパースカラ**(superscalar)と呼ぶ．スーパースカラでは並列に実行できる命令数をウェイと呼ぶ．イシュー(issue：発行)と呼ぶ場合もある．厳密には命令デコーダから複数存在する命令実行パイプラインに同時に送り込む(発行)ことのできる命令数がイシューであり，命令実行パイプラインの本数がウェイである．しかし，現在ではそれほど厳密には区別されていない．どちらかといえば，ウェイという表現のほうがよく使われる．

一般に，2ウェイスーパースカラといえば2命令を並列実行できるパイプライン構造のことである．しかし，アウトオブオーダ実行が当然のようになっている現在の技術では，複数存在する演算器に対して2命令を同時発行できるパイプライン構造(2イシュー)のイメージのほうが強い．

いずれにしろ，スーパースカラの概念は**図1**のようなパイプラインの図で表されることが多い．つまり，命令フェッチ，命令デコード，実行，メモリアクセス，ライトバックを2命令並行に処理する，というイメージである．実際の動作とはあまり一致していないが，直感的ではある．

連続する命令は互いに独立ではなく，相互に関係がある場合がある．このため，単純に命令の並列実行はできない．因果律が逆転するからだ．スーパースカラの最大の特徴は，MPUが複数の命令を並列に実行するからといって，プログラムで特別な考慮をする必要がない点である．従来からの命令セットを変更する必要もない．MPU自身が命令間の依存性を検出し，並列に実行可能な命令を自動的に判定し，演算器に対して発行する．そして各演算器は命令を並列に実行する．

もっとも，命令間に依存関係があると，処理にオーバヘッドが生じ，実行効率が低下するので，スーパースカラの真の性能を発揮するには，プログラム側での考慮(コンパイラによる命令の並び替え)が必要である．このため，新しいMPUが発表になるとき，従来のオブジェクトコードそのものではそこそこしか速くならないが，新しいMPU用に開発されたコンパイラで再コンパイルすると性能が劇的に向上する，ということがよくいわれる．

Column スーパースカラという名前の由来

ここでスーパースカラの名前の由来に触れておこう．スカラから連想されるのは，ベクトル量に対するスカラ量である．つまり，科学計算でおなじみのベクトルや行列演算に特化した並列処理ではなく，スカラ量に対する並列処理という意味でスーパースカラと呼ぶという説が有力である．この意味で，通常のシングルパイプラインをスカラパイプラインと呼ぶこともある．

また，スーパースケーラという呼び方もある．これは，1クロックで1命令を実行するという直感的な基準(スケーラ)を超えるという意味からきているらしい．この説はあまり聞いたことはないが，技術解説で有名な某誌ではそう説明されている．

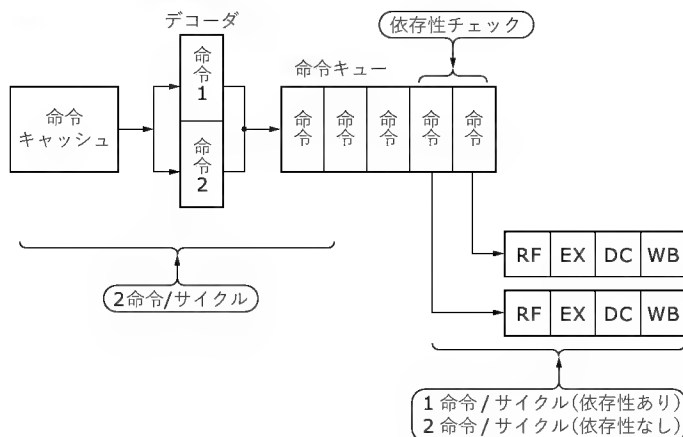
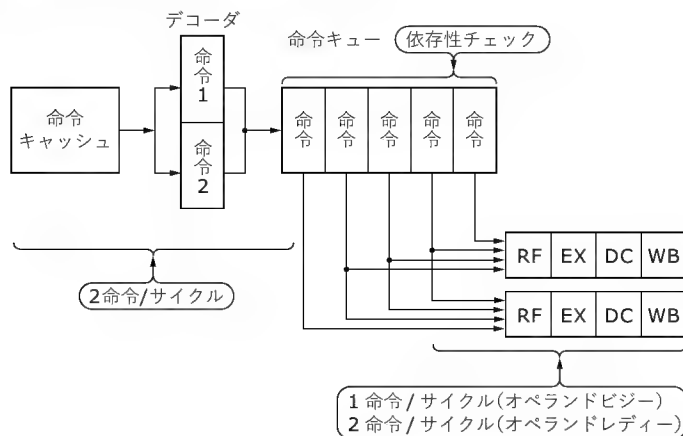
とどのつまり，スーパースカラの語源ははっきりしない．ただ，最近の論文ではスーパースカラの反意語としてユニスカラ(uniscalar)が使用されるが，これなどは「スカラ＝パイプラインの本数」という概念からであろう．

スカラかスケーラかというのは，個人的には単なる発音の問題だと思う．英語による発音はスーパースケーラに近い(少なくともあのHennessy教授はそう発音していた)のだが，最近ではスーパースカラと表記されるほうが多いように思う．実際にスーパースカラと発音する外国人も多くなった．

3 スーパースカラの実現

一般に，命令はデコーダでの発行，演算器での実行，実行の完了の過程を経て処理される．命令の発行はプログラムに書かれた順序で行うこともできるし，矛盾を生じない限りは，プログラムの順序を無視して行うこともできる．また，命令の実行は基本的に1サイクルなので，通常は発行された順序で完了する．ただし，実行クロック数の異なる命令を同時に発行すると，完了する順序が入れ替わることもある．当然，プログラムの順序で完了するとは限らない．処理がプログラムの順序どおりであることをインオーダ，プログラムの順序と異なることをアウトオブオーダと呼ぶ．

スーパースカラの方式は，プログラムの発行，完了が，それぞれ，インオーダかアウトオブオーダであるかによって4種類

〔図2〕インオーダ発行（2ウェイ）

〔図3〕アウトオブオーダ発行（2ウェイ）

に分類できる．以下は簡単のために，2ウェイのスーパースカラを想定して説明する．

● 命令デコード

命令デコードは，インオーダ/アウトオブオーダでそれほど大きな違いはない．命令キャッシュから2命令（たいていの場合ウェイの数と同じ数）をデコードし，命令キューに入れて終了である．デコーダと演算器の中間に命令キューをもつ方式では，デコードと発行を独立に行えるので，（命令キューに空きがある限り）1サイクルごとに2命令をデコードできるので効率がよい．また，逆に考えると，命令キャッシュの参照が少々もたついても，その時間的なロスを命令キューで吸収して見えなくすることが可能である．事実，MIPSのRuby（20Kc）はWay予測を行って命令キャッシュを参照しているが，予測失敗時のペナルティーは命令キューで吸収できると説明している．

命令キューの役割は，デコードと命令実行開始までの待ち時間を最小にする役割もあるが，とくにスーパースカラにおいては命令間のオペランドの依存関係を調べることである．たとえば，片方の（先行する）命令の実行結果を，もう片方の（後続する）命令がソースオペランドとして使用する場合に依存関係があるという．簡単にいうと，レジスタ間のハザードである．もし，2命令間に依存関係がなければ，同時に実行可能なので，演算器に2命令（ウェイの数）を発行する．命令の追い越しを許さない場合（つまりインオーダ）は，デコードしている2命令間でのみ依存性を調べればよいので，いちいち命令をキューに入れなくてもデコーダのみの検査でこと足りる．このため，インオーダなスーパースカラ構造では，命令キューをもたないものも多い．ただし，依存関係がある場合はデコーダで（依存関係が解消するのを）待ち合わせることになるので，各サイクルで，常に2命令をデコードすることはできない．このため，少し効率が悪い．

なお，このような役割をする命令キューは特別にリザベーションステーション（Reservation Station），または集中命令ウインドウ（Central Instruction Window）などと呼ばれる．

また，命令の依存関係は命令キューで解消されているので，いったん演算器で実行が開始されるとレジスタ間のハザードによるストールは発生しない．命令ごとに定められた実行クロック数（レイテンシ）を経て実行が完了する．ただし，データキャッシュアクセスによるストールは発生する可能性がある．多くの場合，データキャッシュにアクセスするためのロード/ストアユニットは実行ユニットと分離されているので，アウトオブオーダの場合は他の命令実行に影響を与えない．インオーダの場合は後続命令が先行するロード/ストア命令を追い越せないので，データキャッシュにアクセス中はパイプラインがストールしてしまう．

ところで，本来RISCは複雑な処理を行う命令を扱わないはずだったが，他社との差別化を進めるうちに複雑な命令も扱うようになってきた．これは，RISCのCISC化に通じる．複雑な命令に関しては，x86プロセッサではマイクロコードで処理するが，たいていのRISCはハードワイヤードロジック（結線論理）で処理する．しかし，これはハードウェアの複雑化を招く．そこで考案されたのが，複雑な命令は複数の単純な命令の組に分割して実行する方法である．これは，x86プロセッサがx86命令をμOPに変換して実行するのと同じ考え方である．たとえば，2001年に発表されたIBMのPower4は，複雑な命令を2命令（Cracking）または3命令以上（Millicode）に分割して命令キューに格納する．そして，分割された命令間で使用できる一時レジスタを4本，プロセッサアーキテクチャのレジスタとは別に備えている．このような傾向は今後増えていくかもしれない．

● インオーダ発行

この場合は，命令キューの先頭の（または現在デコードしている）2命令（ウェイの数）のみの依存性を調べる．命令間に依存性がない限り，2命令（ウェイの数）を同時に発行する．依存性がある場合は1命令のみを発行する．残った命令はその次の命令と組になり，再び依存関係が調べられる（**図2**）．

● アウトオブオーダ発行

この場合は命令キュー全体（あるいは一定の命令数の間）で依

存性が調べられる．オペランドの依存性がない(というか，オペランドをすぐに利用できる)命令のうち，先頭から2命令(ウェイの数)を同時に発行する．発行される順序はプログラムの順序と入れ替わる場合がある．命令キュー内のすべての命令に何らかの依存関係がある場合は，先頭の1命令のみが発行される(**図3**)．

オペランドを利用可能かどうかは，リザベーションステーション内の各命令のソースオペランドとデスティネーションオペランドのレジスタ番号を比較することで検出できる．オペランド間に依存性がある場合でも，デスティネーションオペランドが確定していれば(そのデスティネーションオペランドを有する命令の実行が終了していれば)，ソースオペランドは利用可能と判断する(**図4**)．

● アウトオブオーダ完了

RISCは基本的には命令を1クロックで処理できるが，現実には実行に数クロックかかる命令も存在する．とくに浮動小数点命令は実行に最低でも3クロック程度かかるのが実情である．つまり，MPU内には複数の演算器が存在するが，それらが処理する命令のレイテンシは一般には異なる．ということは，命令がインオーダに発行されようがアウトオブオーダに発行されようが，実行がプログラムの順序で完了する保証はどこにもない．すなわち，スーパースカラではアウトオブオーダ完了が自然な姿である(**図5**)．また，実行が終わった演算器には命令キューから次々と命令を発行すればいいので効率的でもある．

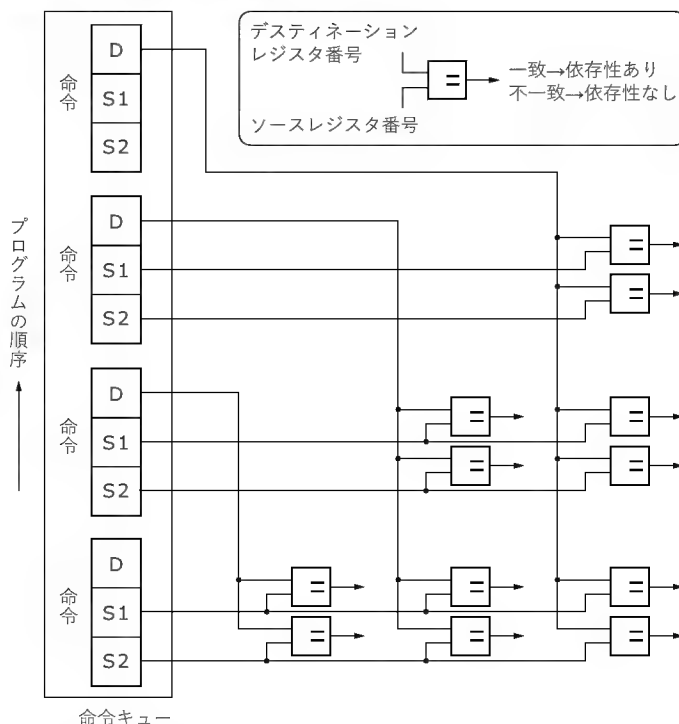
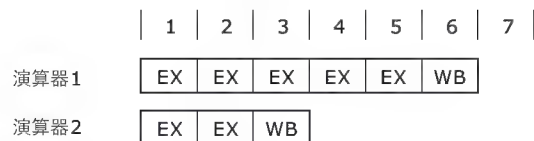
しかし，実行の完了と同時に結果をレジスタファイルに書き戻していたら不都合が生じる場合がある．

まず，第1は出力依存関係である．同時に実行されている命令のデスティネーションレジスタ(結果の格納先)が等しい場合，そこにはプログラム的に後にある命令の実行結果が書き込まれなければならない．ところが，後続命令の処理が先に完了し，先行命令の処理が後から完了する場合，正しい結果(後続命令の結果)が破壊されてしまう．これがWAW(Write After Write)ハザードである．シングルパイプラインではWAWハザードは起こり得ないが，スーパースカラでは当たり前に発生する．もっとも，これはあとで説明するレジスタリネーミングで回避することができる．

しかし，インオーダ発行のスーパースカラでは(回路規模が増大するのを嫌って)レジスタリネーミングを行わないことも多く，この場合はデコード時に発行を待ち合わせたり，後続命令のライトをストールさせるなど，何らかの対策が必要である．

第2は(こちらのほうがもっと深刻だが)，例外の正確性(precise)の問題である．例外はプログラムの順序で処理されなければならない．たとえば，先行する命令も後続する命令も例外を発生する場合，後続命令の例外が先に検出されても，先行命令の例外発生を優先させるようにしなければプログラム処理に矛盾が生じる．

しかし，例外をプログラムの順序で発生させるという制約は

〔図4〕オペランドの依存性チェック

〔図5〕アウトオブオーダ完了(2ウェイ)

演算器1で実行している命令が，演算器2で実行している命令より，プログラム順序で先にあるとき，演算器2の命令で例外が発生すると不都合が生じる

シングルパイプラインでも同様であり，通常なんらかの対策が施されている．問題は例外(割り込みでも同様)発生後に，例外の発生直後の命令からプログラムの処理を再開させる場合(ブレークポイント命令，システムコール命令，トラップ命令，割り込みなどの処理)に生じる．つまり，レジスタへのライトがプログラムの順番を無視して行われていたら，例外からの再開をどの命令から開始してよいのか判断できない．

たとえば，op1，op2を適当な演算として，

R1 ← R2 op1 R3 ……命令1
R3 ← R4 op2 R5 ……命令2
例外/割り込み ……命令3

という命令処理を考え，op2の実行がop1よりも早く終了すると仮定する．この2命令の処理中に後続命令で検知される例外とか割り込みが発生すると，R1は更新されていないのにR3が更新されている状況が発生する．この場合，プログラムの実行再開は，まだ実行されていない命令1から行うことになる．しか

し，命令1のソースオペランドであるR3は命令2で更新される前の値が必要なので矛盾が生じる．例外の正確性を維持するのは，シングルパイプラインではそれほど複雑な制御でないが，(アウトオブオーダ完了の)スーパースカラではかなり複雑である．

例外が発生すると，それは致命的とみなし，プログラムの実行を中断する(再開しない)，割り込みの受け付けは再開に都合のいい時点の処理が終了するまで待ち合わせる，という制御を行えば，アウトオブオーダ完了を実現できる．ただし，システムコールが行えないとか，割り込み応答性が悪くなるという問題が生じ，あまり現実的ではない．

● インオーダ完了

インオーダ完了とは，各演算器の完了がアウトオブオーダに完了するのは避けられないので，その結果を，いったん別の場所に保存しておき，レジスタにライトする順番をプログラムの順番に一致させる方式である．レジスタへのライトが終了するときに初めて命令は真の完了となる．この場合，演算器での実行完了と，命令の真の完了を区別する必要がある．一般には，前者をコンプリート(complete)，後者をリタイアメント(retirement)と呼ぶ．リタイアメントはコミット(commit)と呼ばれることもある．なお，本特集ではリタイアメントという表現は文字数が多いので，その動詞形のリタイアという表現を使用する．

インオーダ完了を実現するために，リオーダバッファ(Reorder Buffer：並び替えバッファ．ROBと省略)という機構が導入される．リオーダバッファとは，プログラムの実行順序を記憶しておくテーブルで，命令の発行時に適当な情報が設定される．その各エントリは，命令がコンプリートしたか否かの情報，命令の実行結果を一時退避するバッファ(このバッファはROBにない場合もある)などからなる．ROBはリザベーションステーションで共用することも可能である．

ROB内にある命令の先頭から，連続してコンプリートしている命令がリタイアできる(**図6**)．1サイクルにリタイアできる最大命令数はMPUごとに異なるが，多くの場合，スーパースカラのウェイ数に等しい．たとえば，命令1，命令2，命令3，命令4がプログラムの順序であり，これらの命令はすべてアウトオブオーダ(である必要もないが)に発行されているものとする．このとき，ROBの内容(先頭4エントリ)が，

命令1　未コンプリート
命令2　コンプリート
命令3　コンプリート
命令4　…

となっている場合，このサイクルでは1命令もリタイアできない．命令1が命令2以降のリタイアを阻害するからである．一方，

命令1　コンプリート
命令2　未コンプリート
命令3　コンプリート
命令4　…

となっている場合は，このサイクルでは命令1のみがリタイアできる．命令3のリタイアは命令2がコンプリートしていないので阻害されている．また，

命令1　コンプリート
命令2　コンプリート
命令3　コンプリート
命令4　未コンプリート

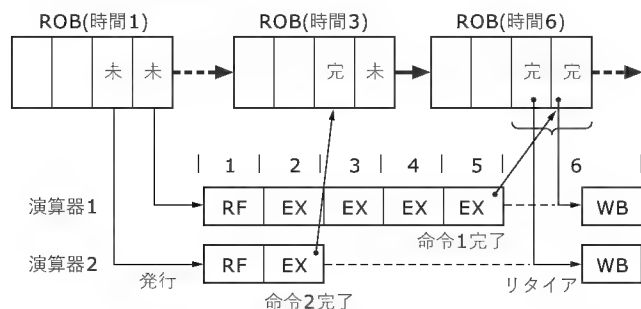
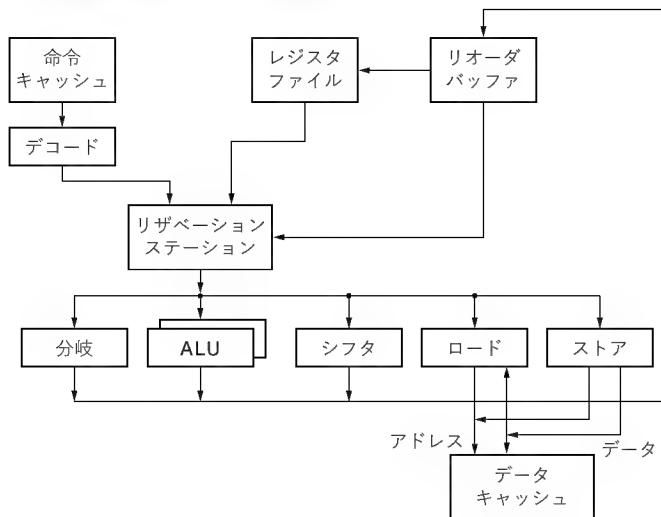
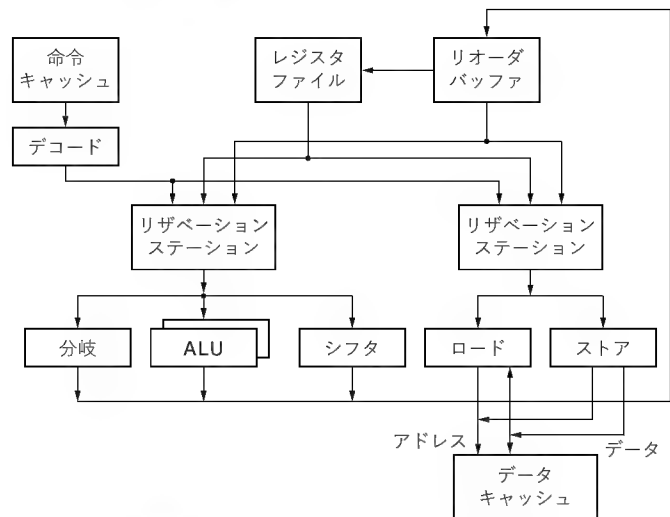
となっている場合は，命令1，命令2，命令3がリタイア対象である．ただし，実際にリタイアできる命令の最大数はMPUごとに異なる．もし，MPUが1サイクルで2命令がリタイア可能なら，命令1，命令2のみがリタイアし，命令3は次のサイクルでのリタイアに回される．もし1度に4命令がリタイア可能なら，命令1，命令2，命令3のすべてがリタイアできる．

例外の正確性の問題があるので，特殊な場合を除き，アウトオブオーダ完了というしくみは採用されない．したがって，スーパースカラの種類は，実質的には，インオーダ(インオーダ発行，インオーダ完了)とアウトオブオーダ(アウトオブオーダ発行，インオーダ完了)の2種類しかない．**図7**に典型的なスーパースカラ構成のMPUのブロック図を示す．**図7(a)**ではリザベーションステーションは一つのみであるが，**図7(b)**のように(いくつかの)演算器ごとにリザベーションステーションを設ける構成もある．

また，インオーダなスーパースカラは2ウェイのものが主流である．インオーダで3～4ウェイというのは記憶にない．これは，多ウェイのスーパースカラ構造を採用する場合でも，整数ALUは2個程度しか用意されていないためではないだろうか．インオーダなスーパースカラではウェイの数だけALUがないとパイプライン効率が悪い．それなら，いっそアウトオブオーダにしたほうが同時発行の効率が上がる．

● 制限付きアウトオブオーダ

複数の演算器それぞれにリザベーションステーション(命令キュー)を有するスーパースカラ〔**図7(b)**のような構成〕において，各命令キューへの命令格納は独立に行われるのが普通である．しかし，すべての命令キューに空きができるまで命令の格納を待ち合わせてから，一括して命令供給を行う方式もある．命令のリタイアもすべての演算器での実行が終了してから一括

〔図6〕インオーダ完了(2ウェイ)

〔図7〕典型的なスーパースカラ構成

(a) 単一のリザベーションステーションを有する場合

(b) 複数のリザベーションステーションを有する場合

して行う．この方式は，演算器の数が少ない場合はインオーダ方式と大差ないので，効率的ではないが，命令キューの待ち合わせ論理を簡略化できる利点がある．

IBMのPower4では，ハードウェアを簡略化して動作周波数を向上させるため，4命令(＋1分岐命令)を一括して，そのグループ単位で実行する方式を採用している．1グループ内の4命令はアウトオブオーダに実行できる．

4 スーパースカラの命令発行を効率的に行うための「レジスタリネーミング」

レジスタリネーミング(Register Renaming)とは，その名称のとおりレジスタ名の付け替えである．その役割には二つある．基本的には，スーパースカラの命令発行を効率的に行うための技術である．

第1は，アーキテクチャ的に定義されたレジスタ数を増やすことである．たとえば，x86の系MPUの汎用レジスタは8本しかないので，ちょっとしたプログラムでもレジスタの使い回しが多くなり，レジスタの依存関係が発生しやすい．これは命令発行の制約となる．レジスタの本数をアーキテクチャが規定するより大きくもち，レジスタの名前を付け替えることで，内部的にプログラムの依存関係を低減できる．

第2は，これも依存関係の解消であるが，WAR(Write After Read)ハザード，WAW(Write After Write)ハザードという偽の依存関係を解消することである．偽の依存関係とは，本来はハザードになるが，レジスタリネーミングによって依存性を解消でき，結果として命令発行の妨げとならないように変更可能な依存関係である．なお，WARとは先行する命令のソースオペランドを後続の命令で変更する可能性のある依存関係，WAWとは後続命令が変更したデスティネーションレジスタを先行する命令が変更する可能性のある依存関係である．これらは同一のレジスタが同時に変更される場合に生じるので，その同一のレジスタを別々のレジスタに割り当ててやれば(偽の)依存関係がなくなる．

また，真の依存関係とはRAW(Read After Write)ハザードのことであり，後続命令が先行する命令の実行結果をソースオペランドとして利用する場合である．これはレジスタリネーミングによっても解消できない．たとえば，次のような命令列を考える．opは単純な加算(＋)よりもレイテンシ(実行時間)の大きい演算とする．

R3 ← R3 op R5 ……命令1
R4 ← R3 ＋ 1 ……命令2
R3 ← R5 ＋ 1 ……命令3
R7 ← R3 op R4 ……命令4

この4命令は(4ウェイスーパースカラで)同時発行しようとしても，命令2と命令3がWARハザードに，命令1と命令3がWAWハザードになっているため，同時発行できない．そこで，次のようにデスティネーションレジスタに対してレジスタリネーミングを行う．基本的には，デスティネーションレジスタを別個のレジスタに割り当てればよい．ソースオペランドは，デスティネーションレジスタの割り当てにしたがって適宜変更される．

P1 ← R3 op R5 ……命令1
P2 ← P1 ＋ 1 ……命令2
P3 ← R5 ＋ 1 ……命令3
P4 ← P3 op P2 ……命令4

このとき，WARハザードとWAWハザードは解消される．しかし，命令1と命令2，命令3と命令4，命令2と命令4は依然としてRAWハザードの関係にある．この場合，依存性のない

命令1と命令3がまずアウトオブオーダ発行できる．命令2は命令1がコンプリートするときに，命令4は命令2と命令3がコンプリートするときに，ソースオペランドが確定するので，晴れて発行できるようになる．

5 分岐予測と投機実行

● 分岐予測

典型的なプログラムでは全体のコードの10％が無条件分岐命令，10～20％が条件分岐命令であるといわれている．無条件分岐は，フェッチするアドレスを分岐先に切り替えるだけなので，それほど問題はない．一方，条件分岐は命令がパイプラインの実行ステージでコンプリートするまで，分岐するか否かが不明なので，やっかいである．分岐命令のコンプリートを待っていたのでは，その間に，多くの命令をフェッチし発行する機会を失うことになる．

そこで考案されたのが，分岐するか否かを推測するアルゴリズムである．もし，推測が成功すれば，命令はほんの少しの遅延（あるいは遅延なし）で続行できる．推測が失敗すれば部分的にコンプリートしている命令を無効化し，正しいアドレスからフェッチ，デコード，発行を再開しなければならない．これは，最近のx86系MPUのようにパイプラインのステージ数が多いMPUではとくに，かなりの性能低下をまねく．しかし，そのようなペナルティを考慮しても，**分岐予測**は必須であり，これを行わないと性能は悲惨なことになる．

分岐予測には二つの基本的な手法がある．静的な分岐予測と動的な分岐予測である．静的な分岐予測は，コンパイラが分岐命令の命令コードに埋め込んだ「ヒント」情報で分岐が発生するか否かを予測する．ただし，このような分岐命令を命令セットとして有するMPUは少ない．あるいは後方（backward）への分岐（オフセットが負）はループの終端とみなせるので，これを分岐すると予測するのも静的な分岐予測といえる．

静的な分岐予測と動的な分岐予測を比較すると，一般には，動的な分岐予測のほうが効果的といわれている．動的な分岐予測とは，分岐命令の時間的な挙動を評価する．一度分岐した分岐命令は次も分岐する傾向があると予測する．これに使われるのは，分岐履歴テーブル（Branch History Table：BHT）と分岐ターゲットバッファ（Branch Target Buffer：BTB）である．BHTもBTBも分岐命令のアドレスをインデクスとするキャッシュである．（キャッシュにヒットする場合）BHTの出力は分岐命令が分岐するか否かの予測情報であり，BTBの出力は予測した分岐先のアドレスである．

● 投機実行 ―― 分岐予測の効果を増大させる

また，分岐予測の効果を増大させるため**投機実行**（Speculative Execution）を行うMPUもある．投機実行とは，分岐予測の成功/失敗がわかる以前でも命令を実行してしまう機能である．しかし，MPUは投機的に実行されている分岐命令の分岐/不分岐が確定するまで（つまり分岐命令のコンプリートまで）リタイアできない．もし分岐予測が失敗すれば，分岐命令以降に実行された命令を放棄して，分岐元から命令の処理をやり直さなければならないからである．投機実行中の命令の結果は，通常リオーダバッファに格納される．分岐予測が失敗したらリオーダバッファの該当エントリを無効化すればよい．

ところで，投機実行に限らず，一般的には演算結果はリオーダバッファに格納されてリタイアを待つが，MIPSのR10000などは（レジスタにネーム後の）物理レジスタを直接更新する．これは，アーキテクチャ上の論理レジスタにも実体があり，物理レジスタは値を一時的な演算結果を保持するものとして区別しているためである．この場合，リオーダバッファの中には演算結果の格納領域は不要である．R10000では命令のリタイア時に，その命令に割りつけられている物理レジスタの値が論理レジスタに転送される．

歴史的にはR10000方式のほうが古く，かつ一般的のように思える．x86系のMPUのように物理レジスタを実質的なレジスタの本数を増加させる目的でレジスタリネームを使用すると，それを一時的な結果の保存場所に利用することはできない．リオーダバッファ内に一時的な結果をもつのは姑息な方法のようにも思える．x86系MPUのシェアは膨大なので，そちらの方式が“大勢”といわれれば確かにそうではあるが．

投機実行を行う場合，分岐条件未確定中のロード/ストアがどのように処理されるかは興味深い．たとえば，ストアを行った後で分岐予測の失敗が判明したとき，キャッシュやメモリに不正なデータが書かれることはないのか不安になる．ロード/ストアがキャッシュ領域に対して行われるものならば，ROBと同等な一時バッファを設けて，ロード/ストア命令のリタイアまで保持すればよい．PentiumではこのバッファをMOB（Memory Order Buffer）と呼んでいる．

ロード/ストアが非キャッシュ領域に対して行われるときは事情が異なる．最近のMPUは専用のI/O命令をもっていないため，メモリ空間にI/Oアドレスを割り付けて，そこを非キャッシュで参照することでI/O機能を実現する（メモリマップトI/O）．I/O装置にはリードを行うと内部状態が変化するものもあり，実際には実行されない（分岐予測が失敗する場合の）投機実行中のロードを行うと周辺が誤動作してしまう．つまり，このような場合，投機実行中の非キャッシュ領域へのロード/ストアは実行してはいけない．また，同様に，非キャッシュ領域へのロード/ストアの順序は変更してはいけない．

最近のMPUは，ノンブロッキングキャッシュ機能を実装し，ロード/ストアもアウトオブオーダに行われるが，これはキャッシュ領域に対する場合のみである．

なかもり・あきら　フリーライタ

Chapter 5

実際のプロセッサではどのように実装されているか

スーパースカラの実際

中森 章

パイプラインやスーパースカラの基本を解説したところで，実際のプロセッサの実装方式を解説する．ここではインオーダ方式の代表例として日立のSH-4とインテルのP5を，アウトオブオーダ方式の代表例としてインテルのP6，MIPSのR10000，モトローラのPowerPC750を紹介する．またPentium4やPentium-M，AMDのAthlon/Hammerについても言及する． （筆者）

はじめに

図1に，レジスタマップ，アウトオブオーダ命令発行，リタイアとリザベーションステーションの関係図（概念図）を示す．これらが，典型的なMPUにおいてどのように実現されているか，その実装方式を見ていこう．

各MPUとも，パイプラインに関しては実行ステージ以降の処理はやや枯れた感じがあり，新規性はない．しかし，命令フェッチ，分岐予測，命令デコードの部分は，各社，いろいろな工夫が見られる．そこを中心に比較するのも興味深い．

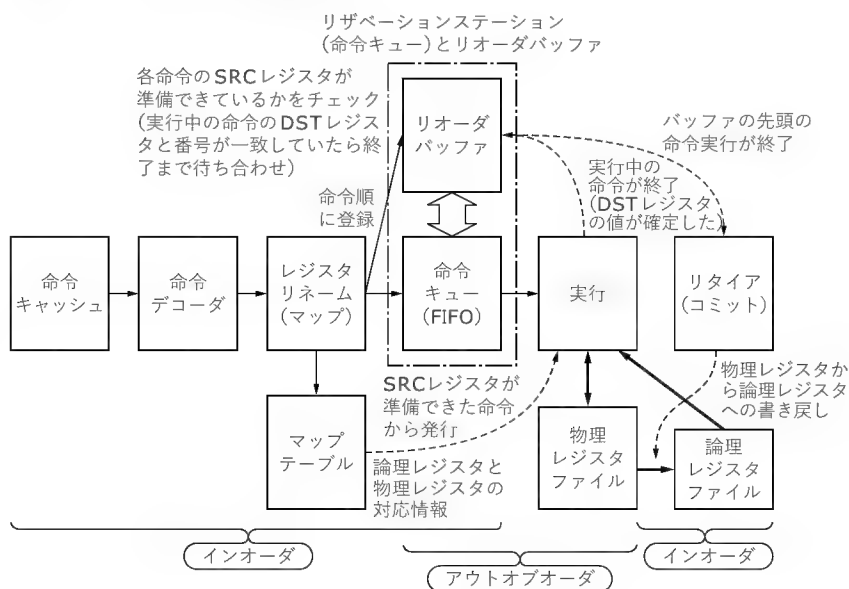
1 SH-4

● インオーダ/2ウェイスーパースカラ構造

SH-4は，整数ユニット，浮動小数点ユニット，ロード/ストアユニット，分岐ユニットという，四つの基本演算ユニットを備える．この四つのユニットに対し，2命令を同時発行するインオーダな2ウェイスーパースカラ構造である．なお，ロード/ストアユニットは単純な整数ユニットの役割をもち，簡単なMOVやNOPなどの命令を0レイテンシで実行できる．

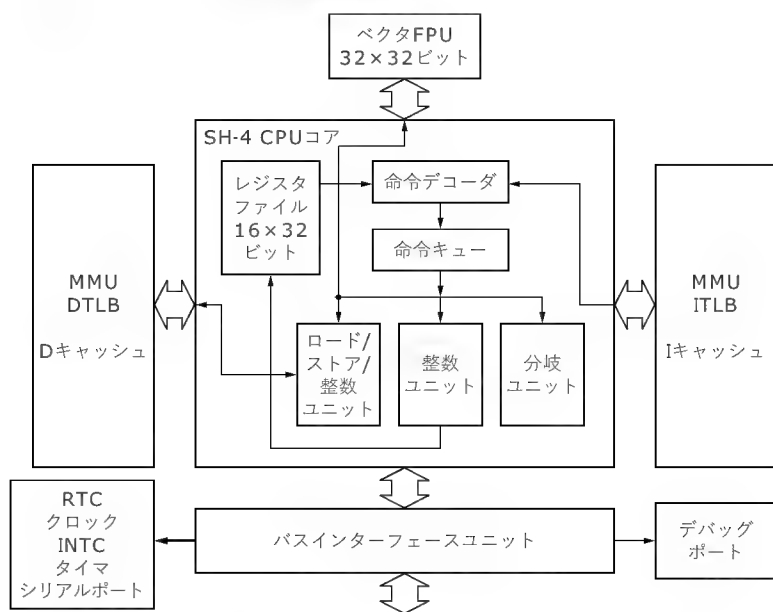
このほかに，複雑な命令を実行するためのユニット（というか上述のユニットを組み合わせて利用？）があるようである．ユーザーズマニュアルによると，SH-4の命令は利用する内部機能ブロックにより，次の6グループに分類できる（略語の意味は筆者の推測によるもの）．

1） MT（Manipulate T）グループ
2） EX（Integer Execution）グループ
3） BR（Branch）グループ
4） LS（Load/Store）グループ
5） FE（Floating Point Execution）グループ
6） CO（Complex）グループ

これらのグループは上述の演算ユニットに対応しており，同時に実行できる組み合わせは**表1**のようになっている．CPUコア内の詳細な内部ブロック図は公開されていない．**図2**は筆者の想像図である．なお，各パイプラインは通常，次の5ステージで処理される．

〔表1〕SH-4の命令の並列実行性

		第2命令					
		MT	EX	BR	LS	FE	CO
第1命令	MT	○	○	○	○	○	×
	EX	○	×	○	○	○	×
	BR	○	○	×	○	○	×
	LS	○	○	○	×	○	×
	FE	○	○	○	○	×	×
	CO	×	×	×	×	×	×

〔図1〕スーパースカラ実現の概念図

〔図2〕SH-4のCPUコアの構造(筆者想像による)

〔図3〕SH-4のパイプラインの命令処理パターン

I ：命令フェッチ
D ：命令デコード
　　発行
　　レジスタリード
　　PC演算(一般命令)
EX ：演算，アドレス計算(ロード/ストア)
SX ：演算，アドレス計算(ロード/ストア)
MA ：メモリデータアクセス
S ：ライトバック
NA ：非メモリデータアクセス
F0/F1/F2/F3：計算
FS ：計算，ライトバック

〔図4〕SH-4の並列動作の組み合わせ

SHAD R0, R1　I D EX NA S
ADD R2, R3　I 　D EX NA S
NEXT　　I D EX NA S

EXグループのSHADと同じEXグループのADDは並列実行できない．SHADのみ発行され，ADDは次の命令と組み合わされて並列実行可能か否かを調べる

(a) 直列実行：並列実行不可能

ADD R2, R1　I D EX NA S
MOV.L @R4, R5　I D EX MA S

EXグループのADDとLSグループのMOVは並列実行できる

(b) 並列実行可能：並列実行可能かつ依存性なし

ADD R2, R1　I D EX NA S
MOV.L @R1, R2　I 　D EX NA S
NEXT　　I D EX NA S

EXグループのADDとLSグループのMOVは並列実行可能．しかし，ADDの結果をアドレスとするので依存性があり，この場合は直列実行になる．MOVは次の命令と組み合わされて並列実行可能か否かを調べる

(c) 並列実行可能：並列実行可能かつ依存性あり

MOV R0, R1　I D EX NA S
ADD R2, R1　I D EX NA S

依存関係がある場合でも，0サイクルレイテンシ命令の後続命令は並列実行できる．R1がフォワーディングされる

(d) 0サイクルレイテンシ命令

MOV.L @R1, R2　I D EX MA S
ADD R0, R2　I D 　EX NA S

MOV.Lは2サイクルレイテンシであり，その結果を後続命令で参照するには，MAステージまでストールする

(e) 2サイクルレイテンシ命令

MOV.L @R1, R2　I D EX MA S
SHAD R2, R3　I D 　d EX NA S

ロード命令の結果をシフト量として使用する場合は，Sステージまでストールする．シフト量にはフォワーディングされる経路がない？

(f) シフト命令のシフト量

1) 命令フェッチ(I)
2) デコード・レジスタリード(D)
3) 実行(EX, SX, F0, F1, F2, F3)
4) データアクセス(MA, NA)
5) ライトバック(S, FS)

これ自体はSH-3のパイプラインと同じである(浮動小数点演算が追加されているが)．**図3**に基本的なパイプラインの命令処理パターンを示す．

ユーザーズマニュアルによると，2命令を同時発行できる場合は，0レイテンシと1レイテンシの命令の組み合わせだけのようである．片方が0レイテンシならば無条件に，1レイテンシ同士なら，命令のグループが異なり，かつレジスタの依存性がない場合に限り同時発行できる．

2命令間が同時発行できないグループにあるとき，実行に必要なハードウェア資源が競合するとき，レジスタの依存関係があるときは同時発行できない．この場合，第2命令は，その後続命令とともに再度組み合わされて，同時発行ができるか否かを決定する．この関係を**図4**に示す．

マニュアルを読み間違えていなければ，マルチステップ命令(この種の命令はハードウェア資源を独り占めする)が出現する場合や，レジスタの依存関係がある場合は，SH-4のパイプライン処理の効率はシングルパイプラインと大差ないように思える．少しの並列実行を行うために，ハードウェア資源をむだ使いしているように感じるのは筆者だけであろうか．せめて，整数演算器がもう一つあれば，性能はもう少し向上するであろう．

〔図5〕**R10000のブロック図**

● 特徴的なSH-4のパイプライン動作例

ここで，SH-4のパイプラインで特徴的な二つの例を示す．

▶0サイクルレイテンシ命令はレジスタの依存関係があってもストールしない

これはソースフォワード機能と呼ばれる(通常のフォワードとは異なる)．おそらく，デコード時に他方の命令のソースオペランドを読み替えるのであろう．たとえば，

```
MOV  R1, R0  // R0 <- R1
ADD  R0, R2  // R2 <- R0+R2
```

という命令列がある場合，ADDのソースであるR0はR1に変換されて，依存性をなくし，この2命令を1サイクルで実行する．

▶条件分岐命令でオフセットが0の場合は分岐成立時でも1サイクルで実行できる

分岐命令が分岐する場合，分岐先のフェッチまでに1サイクルストールするので，レイテンシは2であるが，分岐のオフセットが0(2命令後への分岐)の場合は，分岐先フェッチのオーバヘッドがなく，1サイクルで実行できる．つまり，スキップ命令として利用できる．

たとえば，次のような命令列が考えられる．最近のMPUでも，2命令後への分岐は特別扱いして高速化が試みられているが，これはその先駆けかもしれない(しかし，そのような場合は条件MOVE命令を使用することのほうが多いような気もするが)．

```
CMP/GT R1, R2   // if R2>R1 then set T-bit
BT     label    // branch relative if
                   T-bit set
ADD    #4, R3   // if T-bit set then
                   R3<-R3+4
label:
```

SH-4のパイプラインは2命令同時実行に制限が多い．このあたりはコンパイラのがんばりしだいといえるかもしれない．

日立の発表では，SH-4の性能は200MHz動作で360MIPSということになっている．これはDhrystone1.1による値であり，他社が採用しているDhrystone2.1では300MIPS程度という噂もある(公式発表はない)．200MHzで300MIPSも達成できれば，インオーダスーパースカラとしては，まずまずの性能であろう．しかし，300MIPSというのも少し眉唾な気がしないでもないが(このパイプライン構造のわりに性能が良すぎる)．

● SH-5ではシングルパイプラインへ

SH-4の後継機種である**SH-5**では，スーパースカラ構造では制御が複雑で動作周波数を上げることができないという理由で，7ステージのシングルパイプラインが採用された．これで，動作周波数はSH-4の200MHzから400MHzへと向上するという．しかし，筆者にはSH-4のパイプラインが周波数の向上の妨げになるほど複雑だとは思えない(とはいえ規模は大きそうだが)．むしろ，SH-4ではシングルパイプライン並みの効率のスーパースカラでしかなかったので，効率的なシングルパイプラインで作り直したと考えるほうが納得できる．

実際，SH-5の性能は400MHz動作時に714MIPS(Dhrystone 1.1)と発表されており，IPCで見るとSH-4もSH-5もほとんど同性能と思われる．これが本当なら，SH-5はとても素晴らしいCPUといえる．個人的にはシングルパイプラインでMIPS/MHzの値が1.785というのは不可能だと思うのだが(どう考えても，インオーダスーパースカラの値)．その後，SH-5の性能はDhrystone2.1では604MIPSと発表された．MIPS/MHzは1.51である．これでも性能が良すぎる気がする．こちらもSH-4の性能比(噂であるが)と同じである．

2 R10000

● 4ウェイスーパースカラ構造

次はMIPS系のプロセッサから，**R10000**を取り上げてみる．**図5**にR10000の機能ブロック図を示す．

図を見てわかるようにR10000は，演算器として2個の整数ALU，アドレス計算(ロード/ストア)ユニット，浮動小数点加

算器，浮動小数点乗算器，浮動小数点除算器（平方根器を兼ねる）をもつ．各演算器には，それぞれ16エントリからなる整数キュー，アドレスキュー，浮動小数点キューが接続されている．命令は各キューから対応する演算器に対してアウトオブオーダに発行される．

R10000は1サイクルで，4命令を命令キャッシュからリードし，最大4命令をデコード/レジスタリネーム可能である．デコードされた命令は32エントリのアクティブリストと呼ばれる命令キューと兼用のリオーダバッファに格納され，インオーダなリタイア〔R10000ではグラジュエート（卒業）と呼ぶ〕を管理する．1サイクルで最大4命令をリタイアできる．

演算器は6個備えているが，浮動小数点の乗算器と除算器は入力が共通なので，1度に最大5命令を発行することができる．その意味で5ウェイのスーパースカラであるが，デコード自体は1サイクルで最大4命令なので一般には4ウェイのスーパースカラと呼ばれる．

図6にR10000のパイプラインを示す．一応，7ステージパイプラインの体裁をとっているが，命令のデコード/レジスタリネーム/キューへの格納（ステージ1，ステージ2）までと，発行/実行/結果の格納（ステージ3〜ステージ7）までのステージは独立に動作する分離（decoupled）方式である．アウトオブオーダ方式のスーパースカラとしては珍しくない．

R10000のアーキテクチャをMIPS社は**ANDES**（Architecture with Non-Sequential Dynamic Execution Scheduling）と呼び，アウトオブオーダ実行，分岐予測，投機実行などの総称としている．レジスタリネームは32個の論理レジスタを64個の物理レジスタへ割り当てる．割り当て可能な物理レジスタはフリーリストと呼ばれるテーブルで管理される．物理レジスタは整数と浮動小数点の2系統が用意されている．

分岐予測は，512エントリ×2ビットの情報で過去2回の分岐/不分岐の履歴を保持し，予測した方向に投機実行する．投機実行は，アクティブリストが一杯になるか，レジスタリネームのためのフリーリストが空になるまで，あるいは分岐命令を4個デコードするまで（つまり4回続けて分岐予測するまで）継続される．分岐予測のたびに，その時点での論理レジスタと物理レジスタの対応表（マップテーブル）のコピーを残しておき，分岐予測が失敗すると，そのコピーに基づいてパイプラインを分岐元の状態に戻す．R10000では演算はすべて物理レジスタに対して行われ，アーキテクチャ上のレジスタ（論理レジスタ）にはリタイア時に対応する物理レジスタから書き戻される．

● 命令フェッチ

ここで，R10000のパイプラインの詳細について説明する．

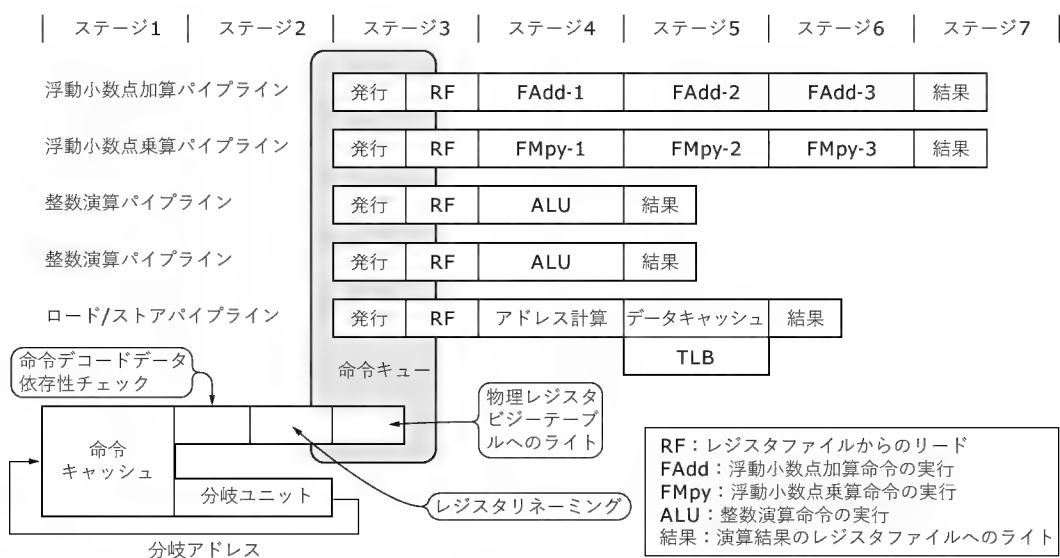
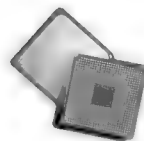
各サイクルにおいて，R10000は32Kバイトの2ウェイセットアソシアティブ構成の命令キャッシュから4命令をフェッチできる．命令キャッシュに格納されている命令は37ビット長で，命令キャッシュのリフィル時に32ビット長の命令をプリデコードしたものである．この余分な5ビットによって各命令を分類し，その命令が実行されるユニット情報を付加することで命令デコードを効率的に行える．このプリフェッチ/プリデコードステージはパイプラインステージの中には数えられない．

● 命令デコード

命令キャッシュからフェッチされた命令は，レイテンシが2ステージの命令デコーダに渡される．実際のデコードは最初のステージで行われ，2番目のステージではレジスタリネームが行われる．

MIPSアーキテクチャではレジスタの本数は整数用と浮動小数点用にそれぞれ32本ある．これは論理レジスタと呼ばれる．R10000はさらにレジスタリネーム用に整数用と浮動小数点用の物理レジスタをそれぞれ64本備えている．プログラム的には32本の（論理）レジスタしか見えてないが，MPU内部では2倍の本数の（物理）レジスタが処理結果を保持している．

レジスタリネームはアウトオブオーダな投機実行を実現する

〔**図6**〕**R10000のパイプライン**

ために重要である．R10000は演算の中間結果や投機実行の結果を，この不可視な物理レジスタに保持する．これらの結果はすべての依存性が解決され，投機的な実行経路が消え去ったときにプログラムから見えるようになる．

MPU内部で何が起きているかを管理するために，R10000はすでに使用されているレジスタ（の番号）を保持するアクティブリストと利用可能なレジスタ（の番号）を保持するフリーリストを用意している．アクティブリスト内のレジスタは二つの状態を取ることができる．一つは「アクティブ」である．つまり，実行中の命令で使用されている状態である．もう一つは「コンプリート」である．つまり，命令実行の最終結果を示す状態である．

ある瞬間には最大32命令が「アクティブ」状態にある．「コンプリート」状態になり結果がグラジュエートすると，不要になったレジスタはアクティブリストから削除されフリーリストに返却される．投機実行はフリーリストに利用可能なレジスタがあり，レジスタリネームが可能な限り継続できる（アクティブリストに空きがあることも必要）．

● 命令実行

R10000は各サイクルにおいて，最大4命令をフェッチし，最大4命令をグラジュエートするが，その中間には五つの実行ユニットがある．このため，可能性としては，各サイクルで，5命令を同時発行，実行，コンプリートできる．このため，R10000は4ウェイスーパースカラとも5ウェイスーパースカラとも呼ばれる．しかし，命令の処理数に関するこの不整合は偶然ではない．ピークのバンド幅を大きくしておくことで，内部資源の割り当てが効率よく行えるし，将来の拡張の余地を残している．……という説明はもっともらしいが，本当にピーク時のバンド幅を考慮するなら，整数演算ALUもFPUも4個ずつ用意すべきであろう．実際，後継機種ではそのような構成を採るという動きもあったようだが，いまだ実現には至っていない．

機能ユニットは二つの64ビット整数演算ALU，一つのロード/ストアユニット，二つのFPUからなる．FPUのうち，一つは加減算用，残りは乗除算/平方根用である．後者のFPUは実際には，同一の発行/コンプリート論理を共有する，乗算，除算（平方根を含む）を行うサブユニットの組である．それらは浮動小数点の乗算と除算を（同時発行はできないが）並行に実行できる．

二つの64ビットALUはほとんど同一である．ただし，乗除算は一方のALUでしか処理できない．他方のユニットには分岐予測の結果を確かめる論理がある．シフトも一方のユニットでしか実行できない．ロード/ストアユニットはすべてのアドレス計算，アドレス変換を処理する．

ここで問題となるのはNOP命令である．MIPSアーキテクチャのNOP命令は「SLL r0, r0, r0」，つまりシフト命令である．プログラム中にかなりの頻度で出現するNOP命令が片方のALUでしか実行できないというのは性能上問題である．これを回避するため，R10000ではNOP命令はプリデコード時に「ADD r0, r0, r0」などの並列実行可能な命令に変換していると聞く．また，これを考慮してか，最新のMIPS32/MIPS64アーキテクチャではSSNOP（SuperScalar NOP）なる命令が定義されている．

五つの実行パイプラインはすべて，最低1ステージからなる実行ステージと，上述のフェッチ，デコード，リネームステージをもち，最後がグラジュエートステージである．このためパイプラインの最小ステージ数は5ステージである．

命令は最初の3ステージを通過するときにはプログラムの順序を維持している．そして，3種類のキューに格納され，最適な実行ユニットに発行されるのを待つ．これらのキュー（ALU, FPU，ロード/ストアユニット用）のそれぞれは16エントリからなり，そのキューのどの位置からでも発行ができる．つまり，この時点からプログラムの順序を維持しなくなる．

ある条件下では，R10000は1サイクルで最大5命令をキューからアウトオブオーダに発行できる．しかし，多くの場合は，命令の依存性に応じて1～4命令を発行する．IPCから察するに，平均は2命令前後であろう．

ロード処理はデータキャッシュにヒットするときに2サイクルかかる．また，ロードは投機的にアウトオブオーダで実行される．これに加え，ノンブロッキングキャッシュ構造によりロードを効率的に処理する．ノンブロッキングとは，ロードがキャッシュにミスしてもストールすることなく先に進める技術である．アウトオブオーダで命令の追い越しが可能な場合，とくに効果的である．R10000では最大四つのロードをノンブロッキングで実行できる．

● グラジュエート（リタイア）

グラジュエートとは物理レジスタの内容を対応する論理レジスタに書き戻す処理である．リタイアとも呼ばれ，インオーダな完了を実現する．

パイプラインの最終ステージにおいて，命令がコンプリートしていても，すべての依存性が解決され，投機的な実行経路が確定するまでは，グラジュエートできない．R10000は正確な例外を保証するので，例外を起こす命令の後続命令はコンプリートしていても，その命令は同様にグラジュエートできない．

グラジュエート時には，物理レジスタが論理レジスタにリネームし直され，その内容が有効になる．このとき，もっとも前にコンプリートした命令から最初にグラジュエートする．グラジュエートを管理するのはアクティブリストである．あるサイクルにおいて，アクティブリストの先頭から見て連続してコンプリートしている命令が，その時点でのグラジュエートの対象になる．したがって，アクティブリストの先頭の命令がコンプリートしない限りは1命令もグラジュエートできない．R10000は各サイクルで，最大4命令をグラジュエートできる．この操作により，命令の流れがその本来のプログラムの順序に戻される．

● R10000の性能

MIPSの発表ではR10000のIPCは1.5だという．これが，Dhrystone MIPSではなく，本来の意味のMIPS値から求めら

れたものとすればかなりの高性能である．出典は失念したが，4ウェイスーパースカラではIPCは1.5程度が限界だそうである．つまり，R10000はIPC的には究極の性能を達成しているといえなくもない．

余談であるが，R10000の後継機種である**R12000**では，動作周波数が200MHzから300MHzに引き上げられたほかに，マイクロアーキテクチャ的には，アクティブリストが48エントリに，分岐予測テーブルが2048エントリに増加している．また新たに32エントリの分岐ターゲットバッファが追加された．R12000の後継機種として**R14000**，**R14000A**，**R16000**が開発されており，その動作周波数は500MHz，600MHz，700MHzである．とくにR16000は2003年4月に発表されたが，1GHzをはるかに下回る周波数は寂しい．

R18000は2001年のHotChipsシンポジウムで概要が発表された．仮想アドレス空間の拡張（52ビット），2個のFPUを実装，L2キャッシュ（1Mバイト）の内蔵とL3キャッシュ（最大64Mバイト）インターフェースの内蔵という点が新たに公開された．動作周波数は800MHz～1GHzと予想されている．周波数的には時代遅れの感がなきにしもあらずである．

3 PowerPC750

● パイプライン

次はPowerPCアーキテクチャのプロセッサを取り上げてみ

Column

V_R4131のパイプライン

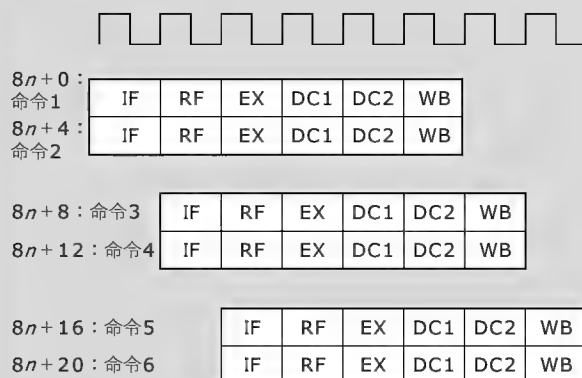
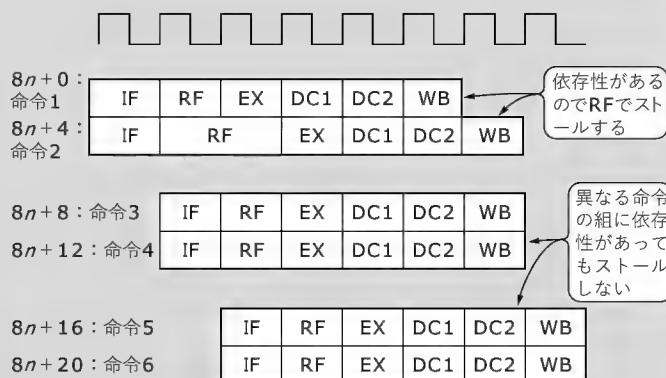
V_R4131はNECが開発した64ビットMIPSプロセッサである．同社の組み込み制御用MIPS RISCとしては初めてスーパースカラ構造を採用した．V_R4131の特徴は，性能もさることながら，低消費電力である点である．通常のスーパースカラ構造では消費電力が上がってしまうため，ユニークなパイプライン構造を採用している．V_R4131では8n番地と(8n + 4)番地の命令が必ず組で処理される．このため，スーパースカラと呼ぶよりもVLIWと呼ぶほうがすっきりする．VLIW構造を採用することで制御回路を単純化し，低消費電力を図っているのである．これは，TransmetaのCrusoeの主張に近い．

V_R4131のパイプラインは典型的な次の6ステージからなる．

1）IF　：命令フェッチ
2）RF　：レジスタフェッチ/命令デコード
3）EX　：実行
4）DC1：データキャッシュ（その1）
5）DC2：データキャッシュ（その2）
6）WB　：ライトバック

このパイプラインを一言で表せば「隣接する2命令単位で処理するシングルパイプライン」である．MIPSの命令は4バイト固定長なので，8n番地と(8n + 4)番地の命令を一つの命令とみなし，それを6ステージのシングルパイプラインで実行する〔**図A(a)**〕．そのためにすべての演算器を2系統備えている（乗除算とロード/ストアユニットは例外）．

隣接する2命令を同時に実行するわけであるが，その命令間にレジスタのRAW依存性がある場合は，当然不都合が起きる．それを回避するために，V_R4131ではレジスタのRAW依存がある場合は，(8n + 4)番地の命令をRFステージで1クロック待ち合わせて，8n番地の演算結果をEXステージからフォワーディングする．つまり，2命令を，依存性がない場合は1クロックかけて実行し，依存性がある場合は2クロックかけて実行する〔**図A(b)**〕．また，実行ユニットを一つしか備えない乗除算命令やロード/ストア命令が隣接する場合は，強制的に依存関係を発生させて逐次的に実行させると思われる．

〔**図A**〕V_R4131のパイプライン

(a) V_R4131のパイプライン（依存性なし）

(b) V_R4131のパイプライン（依存性あり）

よう．PowerPCはIBMとモトローラが独自に開発しているMPUであり，その種類も豊富である．しかし，その内部構造はどれも似ている．

ここでは，任天堂のGAMECUBEのMPUであるGekkoの基になった**PowerPC750**に関して説明する．GekkoはIBMが開発したMPUで，PowerPC750（以降PPC750と略）を下敷きに単精度FPUの強化とL2キャッシュを内蔵したものである．

図7にPPC750の内部ブロック図を示す．また，命令の流れとパイプラインは，それぞれ，**図8**，**図9**のようになる．以下にそれぞれのユニットの動作について説明する．なお，**図9**の各ステージの説明は次のとおりである．

▶フェッチ

フェッチステージは命令が要求されてから命令キューに格納されるまでの間を示す．レイテンシは可変で，命令がBTIC，内蔵キャッシュ，L2キャッシュ，システムメモリのどこにあるかに依存する．命令キューはIQ0～IQ5の六つのエントリをもっている．

▶ディスパッチエントリ中

命令はIQ0とIQ1からディスパッチされる．ディスパッチは瞬間に行われるので「フェッチステージの最後のサイクル」と「実行ステージの最初のサイクル」の中間の時点を表す事象として記述する．

▶実行

命令によって規定される処理はそれに最適な実行ユニットで行われる．この「実行ステージ」から完了（コンプリーション）キューに入る．

▶コンプリート

命令はコンプリーションキューにある．PPC750ではコンプリーションキューがリオーダバッファの役割をしている．最終ステージにおいて，実行された命令の結果がライトバックされ，命令はリタイアする．コンプリーションキューはCQ0～CQ5の六つのエントリをもっている．

▶リタイアメントエントリ中

コンプリートした命令はCQ0とCQ1からリタイアできる．ディスパッチと同様に，リタイアはコンプリートステージにおける最後のサイクルの終わりで発生する事象である．

● 命令の流れ

PowerPCにおいて命令は命令ユニットから実行ユニットに流れて処理される．命令ユニットは逐次フェッチ器，6エントリの命令キュー（IQ），ディスパッチユニット，分岐処理ユニット（BPU）からなる．逐次フェッチ器とBPUから供給される情報に基づいてフェッチすべき次のアドレスが決定される．そして，逐次フェッチ器は命令キャッシュから命令を読み出して命令キューに渡す．BPUは逐次フェッチ器に読み込まれる分岐命令を解析し，分岐条件が確定している場合は分岐命令を削除してフェッチを分岐先に切り替える．分岐条件が確定していない場合は，静的または動的な分岐予測を行い，投機実行をする．分岐条件が確定する間にフェッチした命令はBPUに保持される．

図7や**図8**に示すように，PPC750では命令キューとリザベーションステーションが分離されている．命令キューが，いわゆる通常のリザベーションステーションにあたり，PPC750のリザベーションステーションは，後続命令を時間的なロスなしに実行するための，単なるバッファの役割しかしていない（ように思える）．

命令はインオーダに各実行ユニットのリザベーションステーションに渡されて逐次的に実行される．各実行ユニットの処理は独立なので，全体としてはアウトオブオーダ実行になる．命令はコンプリーションキュー（コンプリーションユニットのリオーダバッファのこと）の管理の下，インオーダに終了する．

命令の処理はリネームされたレジスタに対して行われ，コンプリーションユニットによるリタイア時にアーキテクチャ上のレジスタ（GPR，FPR）に書き戻される．

● 命令キューとディスパッチユニット

命令キュー（IQ）は最大6命令を格納できる．命令フェッチ器はIQに空きができると直ちに命令キャッシュから命令を読み込む．そして，分岐命令を除くすべての命令は，1クロックに最大2命令の割合で，IQの先頭とその次（IQ0，IQ1）から各実行ユニットにディスパッチされる．五つの実行ユニット（IU1，IU2，FPU，LSU，SRU）はそれぞれ独立にリザベーションステーションをもっている．ディスパッチユニットはソースとデスティネーションレジスタの依存性をチェックし，コンプリーションキューに空きがあれば新しい命令を，その命令が実行されるべき実行ユニットに，ディスパッチする．これは，各ユニットのリザベーションステーションに命令を移動することに等しい．

● 分岐処理ユニット（BPU）

BPUは，逐次フェッチ器から分岐命令を受け取り，条件分岐に関するCR（Condition Register）の先読み操作を行うことで分岐条件を早期に確定する．これにより，多くの場合，0サイクルで分岐を実現できる．無条件分岐と条件が確定している条件分岐はただちに実行される．

分岐条件が未確定な条件分岐命令に関しては，アーキテクチャ的に定義された静的な分岐予測または動的な分岐予測を用いて分岐の経路が予測される．予測により，予測した経路から，命令フェッチ，ディスパッチ，実行が行われる．しかし，予測が正しいと決定（解決）されるまで，それまでに処理した命令はコンプリートすることができず，結果をレジスタにライトバックすることもできない．

もし，予測を誤った場合，誤った経路の命令はプロセッサからフラッシュされ，正しい経路からの処理が開始される．いわゆる投機実行であるが，PPC750は2番目の分岐命令まで予測することができる．2番目に予測された命令の経路からは，命令のフェッチはできるがディスパッチはできない．

〔図7〕PowerPC750のブロック図

128ビット（4命令）
タグ
32Kバイト命令キャッシュ
インストラクションMMU
SRs（シャドウ）
ITLB
IBATアレイ
インストラクションユニット
分岐処理ユニット
BTIC 64エントリ
BHT
CTR
LR
フェッチャ
インストラクションキュー（6ワード）
ディスパッチユニット
64ビット（2命令）
2命令
リザベーションステーション
整数演算ユニット1
+ × ÷
整数演算ユニット2
+
システムレジスタユニット
CR
32ビット
GPRファイル
リネームバッファ（6）
リザベーションステーション（2エントリ）
ロードストアユニット
+
（EAカリキュレーション）
ストアキュー
64ビット
FPRファイル
リネームバッファ（6）
浮動小数点演算ユニット
+ × ÷
FPSCR
コンプリーションユニット
リオーダバッファ（6エントリ）
データMMU
SRs
DTLB
DBATアレイ
EA
PA
32Kバイトデータキャッシュ
タグ
64ビット
60xバスインターフェースユニット
インストラクションフェッチキュー
L1キャストアウトキュー
データロードキュー
L2バスインターフェースユニット
L2キャストアウトキュー
L2コントローラ
L2CR
L2タグ
32ビットアドレスバス
64ビットデータバス
17ビットL2アドレスバス
64ビットL2データバス

Information ―マピオンのWindows版IE機能拡張プラグイン ツールバーをリニューアル
（株）サイバーマップ・ジャパンは，同社のインターネット地図サイト「マピオン」において配布中のWindows版IE機能拡張プラグイン「マピオンツールバー」をリニューアルした．このリニューアルは（株）アクセスポートの開発協力によるもの．

〔図8〕PowerPC750のデータの流れ

動的な分岐予測には512エントリの分岐履歴テーブル(BHT)を用いる．これは各エントリに2ビットの情報を保持し，それらは，分岐命令に対する「NOT TAKEN」，「強いNOT TAKEN」，「TAKEN」，「強いTAKEN」という四つのレベルを示す．もし，動的な分岐予測ができない場合は，条件分岐命令に埋め込まれている予測ビットを使用する．

このように，PPC750は未解決の条件分岐命令に出会うと，分岐が解決するまで結果をアーキテクチャ上のレジスタにライトバックはしない(リネームレジスタに保持する)が，予測した分岐先からの経路からの命令を実行する(投機実行)．この実行は2番目の未解決な分岐命令に出会うまで継続する．もし，分岐がTAKEN(TAKENと予測)するとき，TAKENしない経路からの命令は破棄され，予測した側の命令の経路がIQにフェッチされる．

BTIC(Branch Target Instruction Cache)は，最近使用された分岐先の2命令を保持する64エントリのキャッシュである．命令フェッチがBTICにヒットするとき，命令は次のクロックサイクルで命令キューに与えられる．これは命令キャッシュからリードするよりも1クロック早い．それに引き続く命令は次のクロックサイクルで命令キャッシュからリードされる．つまり，BTICは誤って命令をディスパッチする回数を減らし，ターゲットストリームからの1クロック早い開始を可能にする．

BPUは分岐のターゲットアドレスを計算する加算器と3個のユーザーが制御するレジスタであるリンクレジスタ(LR)，カウントレジスタ(CTR)，CRを含む．BPUはある種の分岐命令ではサブルーチンコールの復帰位置を計算し，LRレジスタに格納する．さらにLRは分岐コンディショナルリンクレジスタ(bclrx)命令に対するターゲットアドレスを含む．CTRはカウントレジスタに対する条件分岐(bcctrx)命令の分岐ターゲットアドレスを含む．LRやCTRは特殊レジスタ(SPR)であり，その内容はGPR間で転送可能である．BPUはGPR(整数レジスタ)やFPR(浮動小数点レジスタ)と独立の特殊レジスタを使用するため，分岐命令の実行は整数や浮動小数点命令の実行とは独立して行うことができる．

〔図9〕PowerPC750のパイプライン

● コンプリーションユニット

コンプリーションユニットは命令ユニットと密接に連動して動作する．命令はプログラムの順序でフェッチされディスパッチされる．ディスパッチ時に，6エント

リのコンプリーションキューの連続するエントリに，その命令を入れることによってプログラムの順序を保持する．コンプリーションユニットは命令を，ディスパッチから実行を通じて，追跡し，コンプリーションキューの二つの出口（CQ0，CQ1）からプログラムの順序でリタイアさせる．コンプリーションキューに空きができるまで命令を実行ユニットにディスパッチすることはできない．

また，CTRやLRを更新しない分岐命令は命令ストリームから削除され，コンプリーションキューのエントリを占有することはない．CTRやLRを更新する命令は，それらが実行ユニットに発行されないことを除き，非分岐命令と同様にディスパッチされコンプリートする．

命令をコンプリートさせることはアーキテクチャ上のレジスタ（GPRやFPR，LR，CTR）に実行結果をライトバックすることである．インオーダなコンプリーションを行うことにより，PowerPCが分岐予測の誤りを回復するときや例外を発生する場合の正しい動作を保証する．命令がリタイアするとき，その命令はコンプリーションキューから削除される．

4 Pentium

● Uパイプ/Vパイプのスーパースカラ構造

Pentium（P5）のパイプラインは，i486と同様の5ステージから構成される．MMX Pentiumではフェッチステージが1段追加されて6ステージになる．イメージ的にはそのパイプラインが2本並列に動作するインオーダスーパースカラ構造である．二つの汎用整数パイプラインに加えて，パイプライン化されたFPU演算を同時に実行できる．

これら二つの整数パイプラインは，UパイプとVパイプと呼ばれる．Uパイプではすべての命令を実行できる．一方，Vパイプでは単純な命令のみを実行できる．同時発行可能な2命令をデコードしたとき，（プログラムの順番で）先行する命令はUパイプで，後続する命令はVパイプで実行される．イメージ的には，Uパイプが常に動作していて，後続命令が同時実行可能な場合のみVパイプも使用する，といったところか．

図10にPentiumのブロック図を示す．なお，五つのパイプラインステージの内訳は，次のようになっている（**図11**）．

1）PF　：プリフェッチ
2）F　：フェッチ（MMX Pentiumのみ）
3）D1　：命令デコード
4）D2　：アドレス生成
5）EX　：実行（ALU演算とキャッシュアクセス）
6）WB　：ライトバック

● 各ステージについて

PFステージでは命令キャッシュまたはメモリから命令がプリフェッチ（先取り）される．Pentiumでは，従来のi486などとは異なり，キャッシュが命令キャッシュとデータキャッシュに分かれているので，プリフェッチがデータ参照と競合しない．PFステージでは，二つの独立なラインサイズ（16バイト×2）の組み合わせのプリフェッチバッファが分岐ターゲットバッファ（BTB）と結合されて動作する．条件分岐命令に行き当たるまではプリフェッチは逐次的に進む．条件分岐命令がプリフェッチされるとBTBで分岐予測が行われ，片方のプリフェッチバッファは分岐先のプリフェッチに使われる．これにより，分岐予測によるプリフェッチと同時に，本来のプリフェッチを続行できる．MMX Pentiumでは四つの16バイトのプリフェッチバッファで最大四つの独立な命令の流れをプリフェッチ可能らしい．本当なのかと疑ってしまうが，これはユーザーズマニュアルからの受け売りである．

FステージはMMX Pentiumのみに存在する．このステージでは命令の長さをデコードする．これは従来D1ステージで行われていた処理である．プリフィクスのデコードもFステージで行われる．

MMX Pentiumでは，さらに，FステージとD1ステージの間に命令キュー（FIFO）が存在する．FIFOが空のときは，命令は遅延なしでD1ステージに渡される．FIFOは4命令分用意されていて，各サイクルで2命令を格納可能である．FIFOからは2組の命令が引き出されてD1ステージに渡される．FIFOは通常命令で満たされているので，常に2命令を取り出すことが可能で，1サイクルで実行される平均命令数は2に限りなく近づく．FIFOがうまく機能している限りは，命令フェッチとFIFOからの命令の切り出しでストールは生じない．

D1ステージでは連続する2命令を同時にデコードし発行する．同時に発行できる命令の組は次のような関係にあるものである．

a）ハードワイヤード化された単純な命令
b）レジスタの依存性がない
c）ディスプレースメント付きとイミディエートの組でない
d）プリフィクスでない

なお，FステージをもたないPentiumではプリフィクスがある時だけD1ステージを繰り返す．また，プリフィクスは，他の命令と組になることはなく，Uパイプのみで実行される．すべてのプリフィクスが発行されると，ベースとなる命令（プリフィクスが付加されていた命令）は次の命令と同時発行が可能になる場合もある．

D2ステージではメモリオペランドのアドレスを計算する．また，レジスタオペランドをリードする．i486では，ディスプレースメントとイミディエートを同時に含む命令，または，ベースとインデクスを持つ命令はもう1回D2ステージが必要だったが，Pentiumでは不要になった．

EXステージはALU演算とデータキャッシュへのアクセスを行う．ALU演算とデータキャッシュアクセスの両方の処理が必要な場合，このステージではさらに1クロックが必要である．EXステージでは分岐予測の正当性の検証も行う．ただし，V

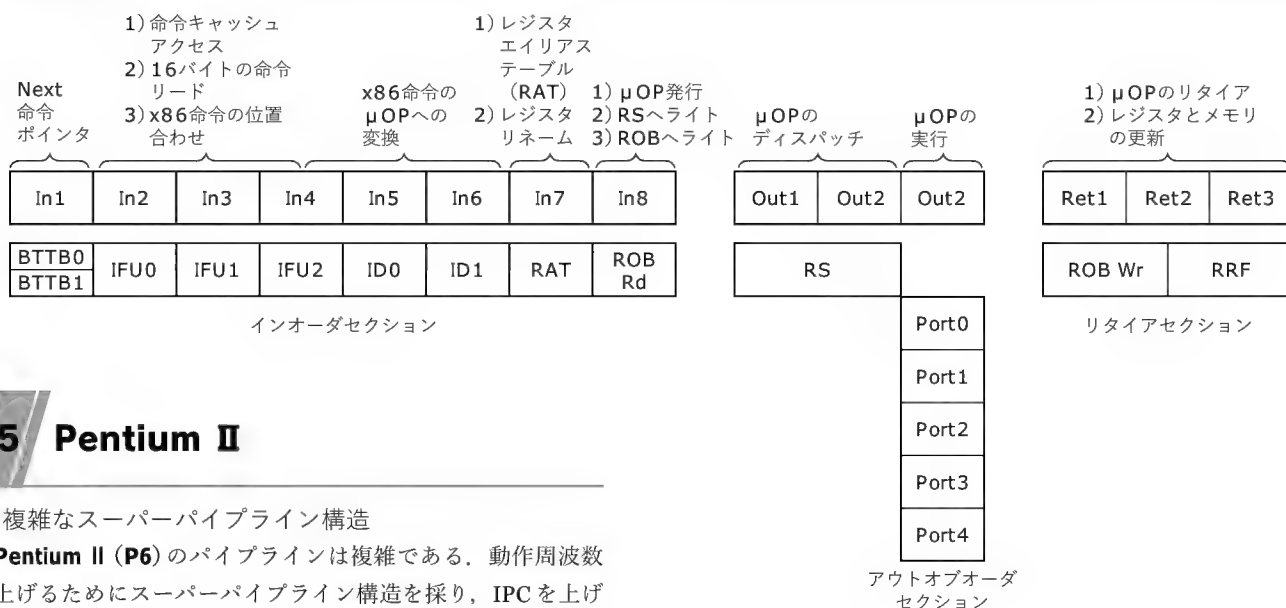
〔図10〕
Pentiumのブロック図

〔図11〕
Pentiumのパイプライン

パイプの条件分岐の検証はWBステージで行われる．また，マイクロコードで実行される複雑な命令はUパイプとVパイプの両方を使う．

WBではプロセッサの状態を更新して実行を完了する．

UパイプとVパイプで実行される命令は同時にD1，D2ステージに入り，同時に抜けていく．当然，EXステージにも同時に入る．もし，片方のパイプがストールすれば他方のパイプもストールする．両方のパイプの命令がWBステージに達するまで，新たな命令はEXステージに入って来られない．こうして，インオーダ完了を実現している．

〔図12〕Pentium IIのパイプライン

5 Pentium II

● 複雑なスーパーパイプライン構造

Pentium II (P6) のパイプラインは複雑である．動作周波数を上げるためにスーパーパイプライン構造を採り，IPCを上げるためにアウトオブオーダなスーパースカラ構造を採っている．14ステージのパイプラインは次の三つのセクションに分割できる．そして，これらのセクションは独立に動作する(**図12**)．

a) インオーダな前処理(8ステージ)

b) アウトオブオーダ実行(3ステージ)

c) インオーダなリタイア(3ステージ)

パイプラインのステージ数は，機能分割のやり方で12ステージ，または10ステージという説もある．ここでは，PentiumIIが発表された当時の一般的な解説記事にしたがう．インテルの公式資料ではステージ数は明記されていなかったと記憶している．マニュアルにある図のステージ数を数えると13ステージのようにも思えるが，実行ステージを抜いて12ステージという解釈が有力であった．なお，Pentium4の発表にあたり，Pentium IIのパイプラインステージ数は公式に10ということになった．

Pentiumは(かなり制限のある)2命令同時発行のMPUだったが，PentiumIIでは3命令同時発行になった．単にパイプラインの本数を増やしただけでなく，PentiumIIはx86命令をμOPというRISC風の固定長命令に変換し，効率的にパイプラインを処理する．x86命令の欠点(?)として，エンコード(命令コードのビット並び)に規則性がないこと，レジスタ-メモリ間演算，可変長命令などが挙げられるが，従来はこれらの特徴が効率的なスーパースカラ処理の妨げとなっていた．μOPを導入することで命令のデコードが容易になり，RISC並みのパイプライン効率を得ることができる．

図13にPentiumIIの機能ブロックを示す．この図を基に命令処理の過程を説明する．

● x86命令の変換

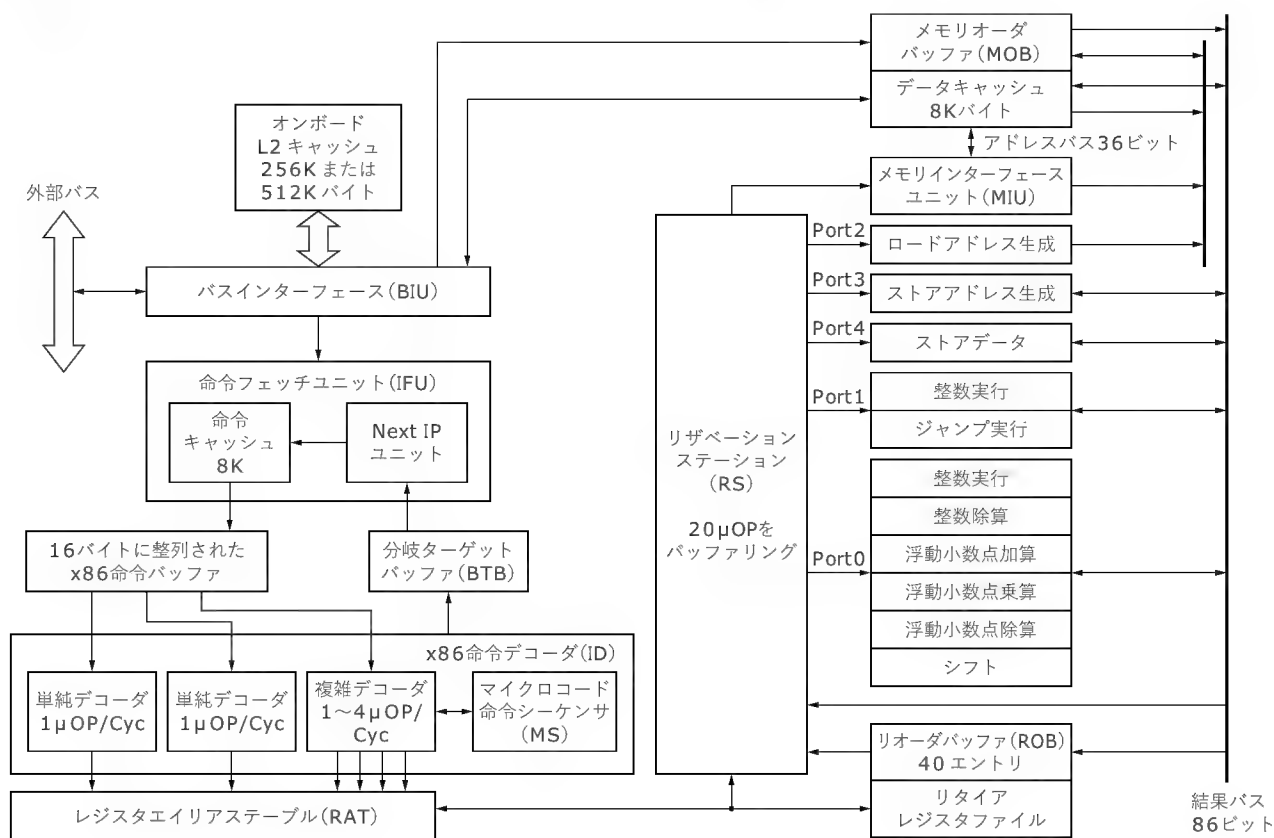
x86命令からμOPへの変換は，パイプラインの最初の8ステージで行われる．まず，分岐ターゲットバッファ(BTB)が指し示す位置の64バイト(キャッシュ2ライン分)のコードを命令キャッシュから読み込む．その中で，最初にあるx86命令の先頭から16バイトのコードを取り出して，並列動作する三つのデコーダに渡す．x86アーキテクチャは可変長命令を採用し，プリフィクスを付加することで(理論上)無限長の命令を生成することができる．

PentiumIIは，1命令の長さを平均5バイトと仮定しているのであろう．もっとも，16バイトのコードのうち，この時点で命令の切れ目は不明なので，三つの命令デコーダは16バイトすべてを受け取ると推測される．Pentium系のMPUでは，命令が16バイト境界にまたがる場合は命令の実行効率が落ちるといわれているが，それはここに原因があると思われる．

さて，PentiumIIの命令キャッシュは1ラインが32バイトなので，その中の任意の位置から始まる16バイトのコードを得るために，二つのラインが同時に読み込まれる．そして，これら3種の命令デコーダがx86命令をRISC命令によく似たμOPに変換する．三つのデコーダのうち，二つが単純デコーダ，残りが複雑デコーダである．単純デコーダは一つのx86命令を一つのμOPに変換する．複雑デコーダは一つのx86命令を一つから四つのμOPに変換する．とくに複雑な命令は複雑デコーダでもデコードできず，そこを通過してマイクロコード命令シーケンサ(MIS)に渡される．MISは必要なだけのμOPを生成する．

たまたま複雑な命令が単純デコーダに割り当てられる場合は，そこから複雑デコーダまたはMISに渡される．ここでのデコードの遅れはリザベーションステーションで吸収されるので，命令の実行には影響ない．

単純な命令と複雑な命令のデコーダへの割り当てが完璧な場合は，1サイクルごとに六つのμOPを生成する．しかし，平均的には1サイクルごとに三つのμOPが生成される．これを根拠にインテルはPentiumIIを「3ウェイスーパースカラ」と呼ん

〔図13〕PentiumⅡのブロック図

でいる．

● レジスタリネーム

μOPに変換されたx86命令は，パイプラインの第7ステージでレジスタエイリアステーブル〔Register Alias Table：レジスタ読み替え表(RAT)〕に送られてレジスタリネームが行われる．ここで偽の依存性(WAWハザードなど)を解消する．

x86アーキテクチャは汎用レジスタ(論理レジスタ)が8本しかないので，レジスタの依存関係は生じやすい．それを軽減させるため，PentiumⅡでは40本の物理レジスタをもつ．つまり，PentiumⅡは内部的に40本の汎用レジスタをもっていることになる．

レジスタリネームでは，真の依存性(RAWハザードなど)は解消できない．しかし，PentiumⅡではレジスタのフォワーディングを行うので，そのペナルティを軽減できる．

● アウトオブオーダ実行

レジスタリネームが完了すると，プログラムの順序で，μOPはリオーダバッファ(ROB)に送られると同時にリザベーションステーションにキューイング(待ち行列に入れる)される．これは，デコーダと実行ステージの中間に位置する．リザベーションステーションは最大20個のμOPを蓄えることができ，11個の実行ユニットに対して，1サイクルで最大五つのμOPを発行できる(入力ポートが五つあるため)．もっとも，典型的なx86の命令列では1サイクルに発行できるμOPはたかだか3命令といわれている．

リザベーションステーションは，ソースオペランドが使用可能になったか，実行ユニットが空いたか，依存性が解消できたかを調べて，用意ができたμOPをアウトオブオーダに発行する．アウトオブオーダにコンプリートするμOPの結果は一時的なバッファ(ROBやMOB)に格納され，ROBの状態を参照しながら，プログラムの順序にレジスタやメモリに書き込まれる．

ROBは40エントリからなる254ビット幅のバッファである．254ビットの内訳は，二つのオペランド，実行結果，多くの状態ビットである．ROBには整数と浮動小数点のμOPの両方が格納される．ROBやMOBから取り出す処理はパイプラインのリタイアのステージで行われる．

● リタイア

ROBは実行状態と各μOPの結果を保持する．μOPは先行するμOPがすべてコンプリートしたことがわかって初めてリタイアし，結果をレジスタやメモリに書き込む．この動作を「コミット」ともいう．PentiumⅡでは1サイクルに最大三つのμOPをリタイアできる．これは，デコーダが1サイクルに発行できる平均的なμOPの個数(3命令)と釣り合いが取れている．

オペランドのフォワーディングのために，それぞれの実行ユニットの結果はすべてリザベーションステーションに戻される．

実行ユニットの結果はROBにも戻されて，リタイアの準備ができたか否かを決定する．

レジスタに対する結果はROBに書き込まれるが，メモリに対する結果はメモリオーダバッファ(MOB)に書き込まれ，対応するμOPがリタイアするまで一時的に格納される．メモリライトを生じるμOPがリタイアして初めてMOBはメモリにデータを書き込む．

● 分岐予測は必須

PentiumⅡはパイプラインのステージ数が多いので，分岐予測は必須である．分岐予測を誤った場合のペナルティは4～15サイクルである．これはかなりの性能低下になるので，高度な分岐予測が要求される．

Pentium(P5)と同様，PentiumⅡは分岐ターゲットバッファ(BTB)を採用する．予測方式は分岐履歴ビットによる．一つの分岐先アドレスに対して，過去4回分の履歴を記録しておき，それにしたがって予測する．これは基本的にPentiumと同じで，4回のループならほぼ100％の分岐予測が可能だという．BTBにヒットしない分岐命令はオフセットの正負などから静的な分岐予測を行う．インテルの主張では，分岐予測の正確さは，Pentiumが80％だったのに対して，PentiumⅡは90％だという．逆にいえば，分岐予測を誤る確率は20％から10％へと半分になったということである(数字のマジック？)．

PentiumⅡで採用している分岐予測の方式は2レベル適応履歴アルゴリズムというものだが，詳細は明らかにされていない．ただし，命令キャッシュのラインごとに四つの分岐先アドレスをBTBで予測することが公表されている．

● 投機実行

PentiumⅡも，分岐予測を有効に活用するために投機実行を行う．PentiumⅡでは，分岐予測が失敗した場合の回復処理は，投機的に実行された命令に対するROBのエントリを無効化することで実現している．PentiumⅡでは他の多くのMPUと同様に，一つ以上の分岐の方向を予測し実行していくという，多重レベルの投機実行を許している．ただし，ROBが一気に無効化されるため，分岐予測失敗時のペナルティは非常に大きい．

PentiumⅡではサブルーチンに対するCALL/RETの組を高速に実行する機構をもっている．サブルーチンはプログラムのさまざまな場所から呼ばれるので，RET命令の分岐先を予測するのは難しい．PentiumⅡではリターンスタックと呼ばれる機構でRET命令の分岐先を予測する．これはCALL命令のデコード時に戻りアドレスを格納するスタックである．RET命令をデコードするとリターンスタックにあるアドレスから分岐先を取り出して，そのアドレスを予測したアドレスとして命令フェッチする．物理的なスタックの内容は他の命令で変更されるおそれがあるので，リターンスタックのアドレスはあくまでも予測値でしかないことに注意すること．なお，これはスタックキャッシュとして昔から知られている手法である．

〔図14〕Penitum4のブロック図

6 Pentium4

● ハイパーパイプライン

インテルの開発した**Pentium4**(コード名Willamette)は，PeitiumⅢに続く製品として2000年に登場した．2GHz以上の動作周波数をめざし，従来の倍のパイプラインステージを採用したため，同一動作周波数ではPentiumⅢよりも性能が劣る．このような状況は，アーキテクチャの変更時には多々あることであり，避けて通れない道である．しかし，何だかんだいっても，現在のIA-32プロセッサの主流はPentium4になっている．

Pentium4のマイクロアーキテクチャは，「NetBurst」と呼ばれる．ここでは，NetBurstのパイプラインの概要について述べる．**図14**にPenitum4のブロック図を示す．Pentium4のパイプラインは次の三つの部分から構成される．これは，PentiumⅡのパイプラインと同様である．

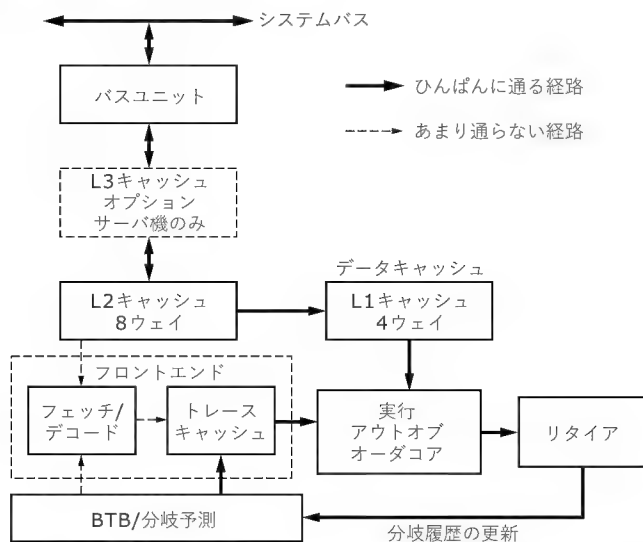
▶インオーダな発行を行うフロントエンド

▶アウトオブオーダなスーパースカラ実行コア

▶インオーダなリタイアユニット

フロントエンドはプログラム順の命令をアウトオブオーダな実行コアに供給する．つまり，IA-32命令をフェッチし，デコードし，マイクロ操作(μOP)に変換する．フロントエンドの主要な仕事は，μOPの連続的な流れを本来のプログラムの実行順序で実行コアに供給することである．

実行コアは1クロックに複数のμOPを発行し，そのμOPの入力の準備ができて，実行に必要なハードウェア資源が利用可能なものから，μOPの順序を入れ替えて実行する．

リタイア部はμOPの実行結果が本来のプログラムの順序にしたがって処理されることを保証し，必要なアーキテクチャ上の状態を更新する．

〔図15〕Pentium4のパイプライン

Prefetch	Decode	Decode	Execute	Wrtback

(a) P5のパイプライン

Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Execute

(b) P6のパイプライン

TC Next IP	TC Next IP	TC Fetch	TC Fetch	Drive	Allocate	Rename	Rename	Queue	Schedule
Schedule	Schedule	Dispatch	Dispatch	Reg File	Reg File	Execute	Flags	Br Check	Drive

(c) NetBurstのパイプライン

● パイプラインステージ数

Pentium4の発表にともない，インテルからNetBurstのパイプラインが公表された（**図15**）．インテルの公式見解では，Pentiumのパイプラインは5ステージ，PentiumⅡのパイプラインは10ステージ，NetBurstは20ステージということになったようだ．20というステージ数は従来のスーパーパイプラインを超えるという意味で「ハイパーパイプライン」と呼ばれる．各ステージの具体的な動作に関しては公式な説明がない．ステージの名称から推測するしかないが，けっこう複雑なことをやっているような気がする．

パイプラインのステージ数が増えた理由は，動作周波数を向上させるためである．Pentiumが233MHz動作，PentiumⅡが1GHz程度の動作であるのに対し，NetBurstでは1.4GHz動作をはじめとして3GHz以上の動作を達成できるといわれている．Pentium4（NetBurst）は，動作周波数が2GHzを超えて初めて存在価値がでてくる．

ところで，NetBurstのIPCが低いということは，パイプラインがスカスカであることを意味する．これはHyperThreadingを実現するためという意見もあるが，真偽はわからない．

● 二つの部分からなるフロントエンド

フロントエンドは，二つの部分からなる．すなわち，

- フェッチ/デコードユニット
- 実行トレースキャッシュ

である．また，フロントエンドは次の基本機能を実行する．

- 実行されると予想されるIA-32命令をプリフェッチする
- プリフェッチされていない命令をフェッチする
- 命令をデコードしμOPに変換する
- 複雑な命令と特殊用途のコードに対しマイクロコードを生成する
- デコード済みの命令を実行トレースキャッシュから供給する
- 高度なアルゴリズムを用いて分岐予測を行う

さらにフロントエンドは，高速なパイプライン処理に関する一般的な問題のいくつかに注目している．たとえば，次の二つの問題に起因する遅延がある．

- 分岐先からフェッチする命令のデコード時間
- キャッシュラインの中間に位置する分岐や分岐先に起因するデコードの負荷

実行トレースキャッシュは，デコードしたIA-32命令を格納することで，これら二つの問題を解決できるように設計されている．命令は変換エンジンによってフェッチされデコードされる．変換エンジンはデコードされた命令を用いて，トレースと呼ばれる一塊のμOPに変換し，実行トレースキャッシュに格納する．実行トレースキャッシュは，これらのμOPをプログラムの実行順序にしたがって格納する．そこでは，コード中に出現する条件分岐の結果（分岐先または分岐元の命令）は予測されて同一のトレースキャッシュのラインに格納される．これにより，分岐によって実行されない命令を格納しないため，キャッシュ容量の効率的な利用が可能になる．あるいは，実行トレースキャッシュは分岐命令をある程度削減しているので，分岐によるペナルティをあらかじめ低減する意味もあると思われる．

実行トレースキャッシュは，実行コアに1クロックに最大三つのμOPを供給できる．この実行トレースキャッシュと変換エンジンは連動する分岐予測ハードウェアと連動している．分岐先はそのリニアアドレスに基づいて予測され，できるだけ早くフェッチされる．分岐先は，もし実行トレースキャッシュにキャッシュされているなら，そこからフェッチされる．もしキャッシュされてない場合は，外のメモリ階層（L2キャッシュなど）からフェッチされる．変換エンジンの分岐予測は実行されると予想される経路にしたがってトレースを形成する．

● 実行トレースキャッシュの構造

さて，インテルが出願している米国特許6,014,742にしたがって，実行トレースキャッシュの構造を予想してみる．トレースキャッシュは**図16**のような構成をしている．ある程度の数の命令（μOP）を実行順に（予測して）並び替えたものがトレースである．

トレースを形成するとき，分岐予測にしたがって動的に実行される命令の流れを追っていくが，それは無限に継続するのではなく，ある程度進んだ時点で中断する．単純には分岐から分岐までを一つのトレースとして形成すればいいのだが，前記の公開特許ではトレース内に条件分岐命令が含まれることを想定

している．

トレースを実際にどの時点で中断するのかはよくわからない．実行トレースキャッシュはこのトレースをキャッシュしたものであり，各トレースは3μOP（公開特許では6μOP）からなるトレースラインから構成されている．実行コアには現在のトレースの中からトレースラインの内容を順次実行コアに送る．

実行トレースキャッシュにはTBTB（Trace Branch Target Buffer）と呼ばれる独立した分岐予測機構がつながり，実行コアに与えるトレースラインごとに分岐予測を行い，トレースラインの供給を継続するか中断するかを決定する．この分岐予測は，トレース形成時の分岐予測とは独立していて，二つの予測が一致するときのみ，トレースラインの供給を継続する．二つの予測が異なる場合は，キャッシュされているトレースに新たな分岐先があるか否かを探し，トレースキャッシュ内にあれば（要はトレースキャッシュにヒットすれば），該当するトレースラインを遅延なしで実行コアに供給する．

実行コアから見れば，分岐予測が非常に正確に行われており，正しい経路の命令が供給されているように見える．新たな分岐先が実行トレースキャッシュになければ（トレースキャッシュミス），トレース単位で不要なものと入れ替えが行われる．命令がループになっている場合は同じトレースに何度もヒットすると考えられる．

〔図16〕Pentium4のトレースキャッシュの構成

だいたいこのような感じであるが，以上は公開特許からの筆者の想像なので，現実の実装と異なっていてもご容赦願いたい．

1トレースラインに3μOPが含まれるということは，12,000命令を格納するというトレースキャッシュは4,000ラインから構成されることになる．このトレースラインをいくつか寄せ集めたものがトレースである．なお，トレースキャッシュの容量は，μOPが12,000命令ということであるから，x86命令に変換すれば16Kバイト相当といわれている．

ところで，実行トレースキャッシュの発想は，TransmetaのVLIWプロセッサであるCrusoeのトレースキャッシュとよく似ている．命令を実行するコアはスーパースカラとVLIWという違いがあるものの，x86命令を実行コアが都合のいい別の形態にプリデコードしてキャッシュする．プリデコード時に分岐命令の挙動を予測し，プログラムの順序ではなく，実行する順序に並び替えてキャッシュするところもそっくりである．この並び替え操作をPentium4はハードウェアで実現するが，CrusoeはCMS（Code Morphing Software）で実現する．

● アウトオブオーダ実行コア

命令をアウトオブオーダに実行するコアの機能は並列性を可能にする主要な要素である．この機能は，一つのμOPがデータや関連する資源を待つ間に待ち合わせが必要なら，プログラムの順序では後に現われる他のμOPを先行して処理させるように，命令の並び替えを可能にする．

プロセッサはμOPの流れをスムーズにするためのいくつかのバッファを備えている．これは，プロセッサのパイプラインの1か所が遅延しても，並行に実行している他の操作や（コアでの効果），先だってバッファにキューイングされているμOPの実行（フロントエンドでの効果）によって，その遅延が埋め合わされることを意味している．

実行コアは並行な実行が可能なように設計されている．四つの発行ポートを通じて，1クロックに最大六つのμOPをディスパッチ可能である．発行ポートを**図17**に示す．1クロックに6命令のμOPの発行することは，トレースキャッシュやリタイアユニットの処理能力を超えていることに注意．これにより，ピークの処理能力を3μOPより大きくし，異なる発行ポートへのμOPの発行に柔軟性を持たせることでより高い発行の割合

〔図17〕Pentium4の実行ユニットとアウトオブオーダコアのポート

を実現している．

ほとんどの実行ユニットは各サイクルで新しいμOPの実行を開始できる．このため，同時に複数の命令が，それぞれのパイプラインで処理状態になる．算術論理演算ユニット（ALU0/ALU1）を用いる多くの命令は1クロックに2命令を開始できる（倍速で動作する）．また，浮動小数点演算命令の多くは2クロックに1命令の割合で開始できる．μOPは，その入力データの用意ができて資源が利用可能になれば直ちに，アウトオブオーダに，実行を開始できる．

● リタイア

リタイア部は，実行されたμOPの結果を実行コアから受け取り，その結果を本来のプログラム順序にしたがって，アーキテクチャ上の状態を正常に更新する．意味的に正しい実行のために，IA-32命令の結果は，リタイアする前に，本来のプログラムの順序でコミットされなければならない．例外は命令がリタイアするときに発生する．例外は投機的には発生せず，正しい順序で発生し，プロセッサは例外処理後に正しい位置から再開される．一つのμOPがコンプリートし，結果をデスティネーションにライトするとリタイアである．1クロックに最大三つのμOPをリタイアできる．

リオーダバッファ（ROB）は，コンプリートしたμOPを格納し，アーキテクチャ上の状態をインオーダに更新し，例外の順序を管理するユニットである．リタイア部は，分岐を追跡し，分岐先の情報を分岐ターゲットバッファ（BTB）に送り，分岐の履歴を更新する．

7 Pentium-M

● x86最新アーキテクチャ

Pentium-M（Banias）に関する情報はほとんど公開されていないが，2003年5月21日発行の*Intel Technology Journal*のVol.7 Issue 2の3番目の記事にPentium-Mのマイクロアーキテクチャの解説がある．これを読んでも，マイクロアーキテクチャに不明な点は多い（パイプラインなど）が，Advanced Branch Prediction（進んだ分岐予測），μOPフュージョン，Dedicated Stack Engine（専用スタックエンジン）についての解説がある．これらについて簡単に解説しておく．

● Advanced Branch Prediction

Pentium-MのマイクロアーキテクチャはPentium IIIに基づいているというのが定説だが，分岐予測に関してはPentium4の技術を採用しているらしい．特殊なプログラムの流れを追っていくため，IP（Instruction Pointer = Program Counter）に基づいて分岐先の成立/不成立を予測する通常の分岐予測機構のほかに，ループ検出器（Loop Detector）と間接分岐予測器（Indirect Branch Predictor）を備える．

ループ検出器は，ループ動作を検出して，その分岐先を予測する．ループは一定の回数同じ方向に分岐し，1回だけ逆方向に分岐する．このため，ループ回数が判明していればループ操作を完全に予測できる（**図18**）．ループが検出されると分岐予測機構の中に1組のカウンタを割り当てて回数を計数していくが，ループ回数をどのように決定するのかは明らかにされていない．

〔図18〕Pentium-Mのループ検出の論理

〔図19〕Pentium-Mの間接分岐予測器の論理

間接分岐予測器は，プログラムの流れによってデータ依存のある間接分岐を解消する．間接分岐は，オブジェクト指向コード（C++やJava）で多用されるが，その分岐予測が誤ると分岐予測性能の低下につながる．

ほとんどの間接分岐は，実行時には，同一の分岐先に分岐する傾向がある．しかし，Javaのバイトコードのインタプリタや C++のCASE文は，データに依存して複数の分岐先をもつ．

間接分岐器は，IPで参照する分岐ターゲットキャッシュと，大域履歴（Global History）でインデクスする分岐ターゲットキャッシュをもつ．IPでの予測が成功した場合はその分岐先を使用し，IPでの予測が外れた場合は大域履歴の分岐先を使用する（**図19**）．

大域履歴は，IPに基づいた予測の付随物であり，IPでの予測が外れると大域履歴に登録する．つまり，過去何回かの間接分岐の分岐先を記憶しておき，もっとも確率の高い分岐先を選択する．

Pentium-Mの分岐予測は前の世代の設計(Pentium IIアーキテクチャ)よりも,予測を外す確率が20%低下しており,実際の性能は7%向上するとしている.この向上率の約30%はループ検出器と間接分岐予測器の組み合わせが寄与している.

● μOPフュージョン

x86プロセッサでは,IA-32の命令(マクロ命令)をμOPと呼ばれるRISC命令に変換して,RISCエンジンで実行することがなかば常識である.しかし,一つのマクロ命令は複数のμOPに分解されるので,リネームやリタイアのバンド幅やリオーダバッファやリザベーションステーションの容量といったハードウェア資源の不足をまねく.これが性能低下に結びつく.それを解消するための手段がμOPフュージョンである.

基本アイデアは複雑な操作(3個以上のオペランドが必要な操作)を行うマクロ命令も一つのμOPとしてデコードして,割り当て,リネーム,リオーダバッファやリザベーションステーションへの登録を行う.これがフュージョン(融合)ということらしい.融合というよりは,分解しないといったほうが正確である.

従来のμOPは2個のオペランドしかもてなかったので,3個以上のオペランドが必要なマクロ命令は2個以上のμOPに分解していた.融合されたμOPをサポートするために,Pentium-Mでは,リザベーションステーションの各エントリは最大3個のソースオペランドを収容できるようになった.また,命令デコーダも,マクロ命令とμOPの対応が1対1になるので,複雑デコーダだけでなく,単純デコーダだけですべての命令デコードが可能になるという.

リザベーションステーションに格納されたμOP命令は,実行ユニットへのディスパッチ時に,本来の2オペランドの複数のμOPに変換される.そして,分解された従来と互換のμOPが実行ユニットでアウトオブオーダに実行され,一つの融合されたμOPを構成する複数のμOP(従来形式)がすべて完了すると,その融合されたμOPがリタイアする(**図19**).

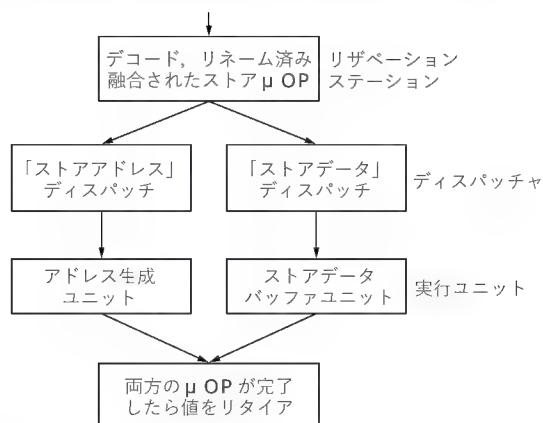
上述の論文では,μOPフュージョンの例として,ストア操作とリードモディファイ(load-and-op,リードした値と演算する)操作が挙げられている.これらのマクロ命令は,ディスパッチ時に2個のμOP(従来形式)に変換される.ストア操作は「ストアアドレス操作」と「ストアデータ操作」に分解される.リードモディファイ操作は「ロード操作」と「演算操作」に分解される.また,本来,マクロ命令が2個のμOPに分解される場合は稀であるとされている.

融合されたストア命令を形成する2個のμOPは並列に発行できる.メモリへの実際のライトはストア命令がリタイアされたときに行われるので,それまでにストアデータバッファに対してアドレスとデータが供給されていればいい.ストアアドレス操作はアドレス生成ユニットへディスパッチされ,そのソースオペランド(ベースやインデクスレジスタ)が用意されたときに実行される.ストアデータ操作はストアデータバッファユニットにディスパッチされ,そのソースオペランド(ストアするデータ)が用意されたときに実行される.これらの実行は独立に行われ,融合されたストア命令のリタイアは両方の操作が完了した時に発生する(**図20**).

融合されたリードモディファイ命令を形成する2個のμOPは,アドレス依存があるため,逐次的に適切な実行ユニットに発行される.ロード操作のディスパッチは,そのソースオペランド

〔図20〕Pentium-MのμOPフュージョンの領域

命令デコーダ
μOPの割り当て,リネーム
μOPのディスパッチ
レジスタファイル μOPのリタイア
融合された μOPの領域
実行ユニット
融合されてない μOPの領域

〔図21〕Pentium-Mの融合されたストアの流れ

〔図22〕Pentium-Mの融合されたリードモディファイの流れ

(ベースやインデクスレジスタ)が用意できたときに実行される．演算操作は，ロードが完了し，もう一方のオペランドの用意ができたときに実行される．融合されたリードモディファイ命令のリタイアは，両方の操作が完了したときに発生する(**図21**)．

上述の論文にはとくに明記されていないが，x86の特徴であるリードモディファイライト命令はリードモディファイ操作とストア操作の組み合わせなので，1個の融合されたμOPとしてリザベーションステーションに登録され，ディスパッチ時に，3個または4個のμOPに分解されるのであろう(メモリのアドレスが同一なのでアドレス計算が1回省略できる)．

インテルによると，融合されたμOP構造はアウトオブオーダロジックで処理されるμOPの数を10％以上減少させることが判明している．μOPの数が減少するため，発行，リネーム，リタイアのスループットが増加し，結果的に性能を増加させる．とくに，命令デコードに監視/複雑デコーダが不要になるため，プロセッサのデコード，割り当て，リタイアのバンド幅を3倍に拡大するとしている．

μOPフュージョンによる性能向上は，典型的には，整数コードでは5％であり，浮動小数点コードでは9％である．ストア操作の融合は，とくに整数コードの性能向上に寄与する．浮動小数点コードの性能向上は，ストア操作とリードモディファイ操作(**図22**)の両方の形式が寄与するという．

● Dedicated Stack Engine

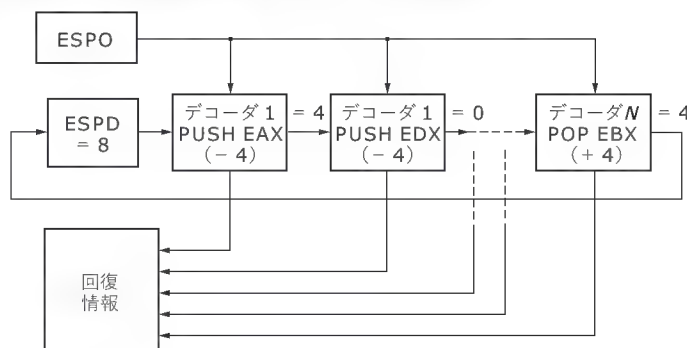
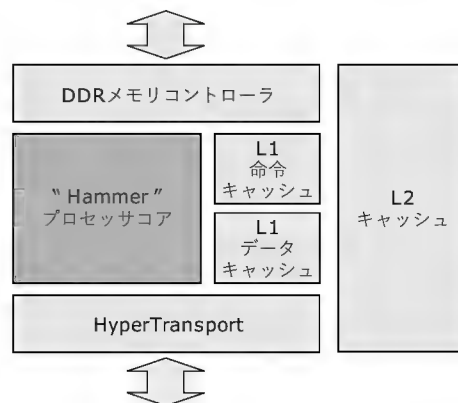
IA32はCISC命令であり，PUSH，POP，CALL，RETなどスタック操作を多用する．これらの命令はスタックポインタ(ESP)の値をアドレスとして使用する．このため，データの移動とは別にスタック計算用のμOPが余分に発行され，μOPの命令数が増加する．また，ESPが更新されないと次のPUSHやPOPが実行できないという依存性も発生する．従来は複数のスタック操作命令を同時にデコードしようとしても，ESPの値が確定していないため，それが不可能だった．Pentium-Mでは，命令デコーダの近くに専用回路を設けることで，非常に効率的に，ESPのこれらの副作用を扱えるようになっている．

その基本原理は，プログラマに見えるESP(ESPP)は，アウトオブオーダ実行エンジンの中にあるESPレジスタ(ESPO)に差分(ESPD)を加えたものである．つまり，

ESPP = ESPO + ESPD

であり，ESPDは命令デコーダで管理できる．つまり，前の命令でのESPの変化量は±4であることが多い(PUSHA，POPAは例外)．命令デコード時にESPDからの変化量を計算し，ESPPの値を推定することで複数命令の同時デコードを可能にするものである(**図23**)．この場合，ESPDの更新は専用の加算器で行う．

この操作は，ESPをデスティネーションオペランドとする命令には効果がない．この場合は，ESPPを計算するμOP(ESPOとESPDを加算)を余分に追加してESPOが更新されるのを待つ．その後はESPDを0とみなしてデコードしてよい．もともとESPDが0の場合は，このような同期化処理は不要である．

〔**図23**〕**Pentium-Mの専用スタックエンジンの論理**

〔**図24**〕**Hammerの構成**

また，Pentium-Mは投機実行を行うので，分岐予測が外れた場合はESPOやESPDの値をある時点まで巻き戻さなければならない．ESPOはアウトオブオーダ実行エンジンのレジスタの一部なので自動的に回復される．ESPDに関しては，その値を保持するテーブルを用意して対応する．

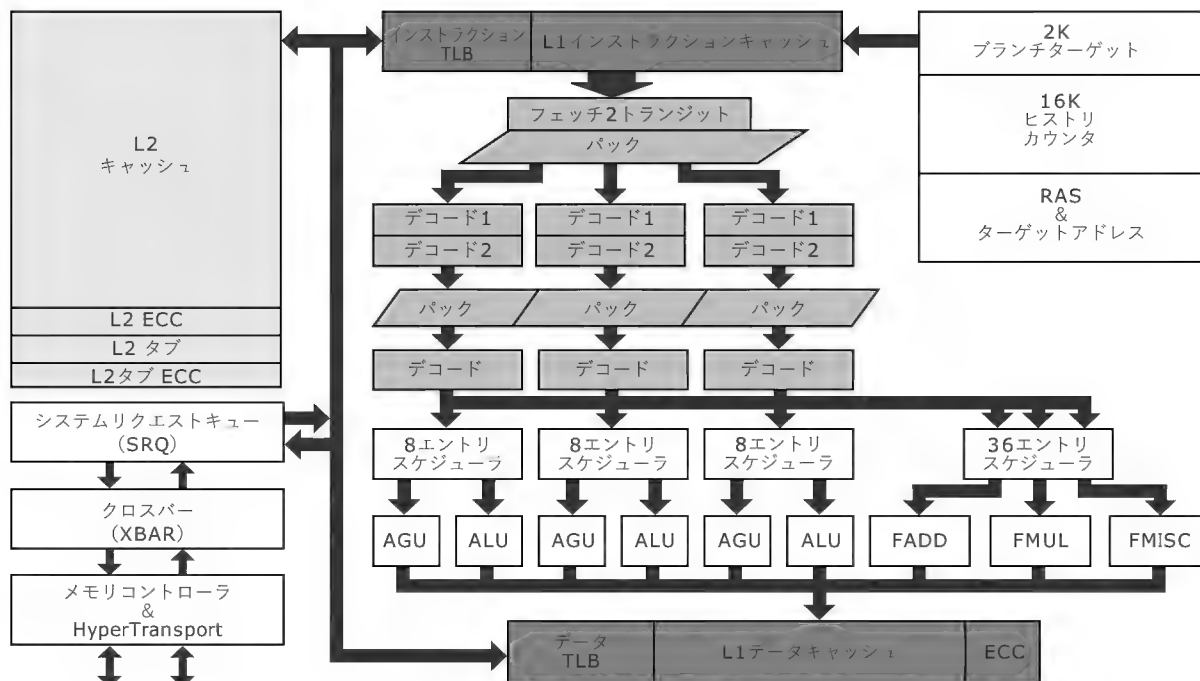
Dedicated Stack Engineを搭載することで，ESPを同期化するμOPを挿入したとしても，μOPの数が5％減少するという．それよりも，命令デコードのバンド幅が向上したことが性能に大きく寄与するらしい．また，消費電力も5％程度の削減になるという．

8 Hammerのパイプライン

● Quanti Speedアーキテクチャを採用

Athlonは，AMDがP6(Pentium II)対抗として開発した32ビットMPUである．そして，**Hammer**はAthlonの後継にあたる64ビットMPUである．マイクロアーキテクチャの基本構造は，HammerとAthlonでよく似ている．しかしHammerでは，I/OやマルチプロセッサACC接続を行うHyperTransportを3ポートとNorth Bridge(DDR SDRAMコントローラ，APICなど)を内蔵している点が異なる．

図24にHammerの構成を示す．キャッシュとCPUコアが分離され，いろいろな構成(高級版や廉価版など)に対応できるようになっている．Hammerアーキテクチャを採用するMPUと

〔図 25〕
Hammer のブロック図

しては，Opteron ブランドの **SledgeHammer**（サーバ，EWS 向け）と Athlon64 ブランドの **CrawHammer**（PC 向け）が発表されている．これらにどのような相違があるのかは不明である．しかし，L2 キャッシュの容量，DRAM コントローラや Hyper Transport のバス幅/ポート数の違いによって区別されると予想されている．

Hammer のブロック図を**図 25** に示す．この図で L1 キャッシュ，L2 キャッシュ，TLB，分岐予測機構，North Bridge 以外の部分がプロセッサコアである．なお，APIC（Advanced Priority Interrupt Controller）とシステム要求キュー（SRQ：System Request Queue）は二つの CPU を扱える構成になっている．当初の発表で Hammer はシングル CPU コアとなっていたが，実際には 2，4，8CPU の CMP（Chip Multi Processor）構成が可能である．

ところで，Hammer は，Athlon と同様に，QuantiSpeed アーキテクチャを採用する．つまり，

- 9 命令同時発行，スーパースカラ，パイプライン化されたマイクロアーキテクチャ
- 複数の並列 x86 命令デコーダ
- パイプライン化された三つのスーパースカラ浮動小数点ユニット（FPU）
- パイプライン化された三つのスーパースカラ整数演算ユニット（ALU）
- パイプライン化された三つのスーパースカラアドレス生成ユニット（AGU）
- 72 エントリの命令制御ユニット
- ハードウェアによるデータのプリフェッチ（Hammer のブロック図にはない）
- 排他的，投機的に入れ替えを行う TLB
- 動的な分岐予測

により，IPC を向上させている．同じ QuantiSpeed アーキテクチャを採用する Athlon と，基本的には変わらない．しかし，整数演算系のスケジューラ（リザベーションステーション）が 18 エントリから 24 エントリに増加している．Athlon のスケジューラの実態は，3 × 6 エントリに分割されているという．その意味で，Hammer では ALU と AGU のペアごとに 2 エントリの増加になっている．

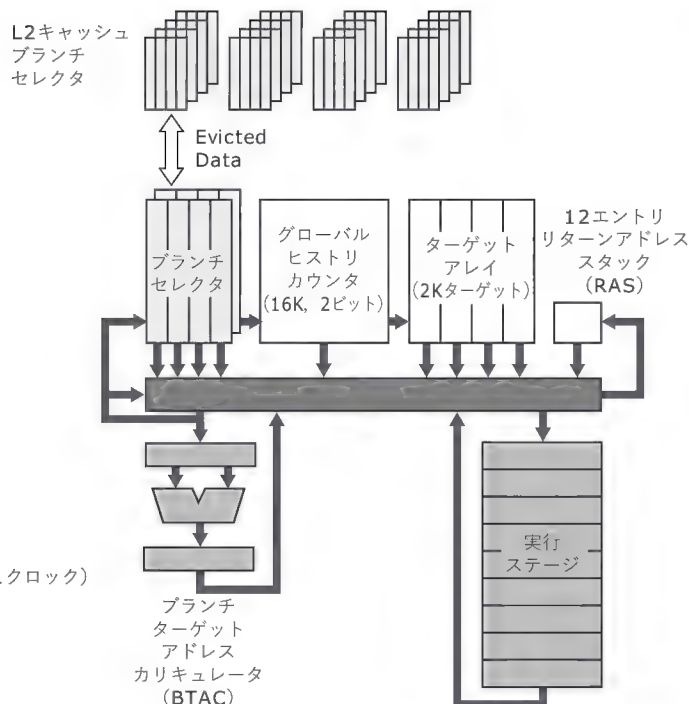
しかし，Athlon と Hammer の大きな違いは，命令のフェッチ系にある．つまり，分岐予測機能の高度化とマルチプロセッサで高性能をねらっている（動作周波数の向上は当然）．命令デコーダ自体に大きな変更はないようである．

● 強化された命令デコーダ

QuantiSpeed アーキテクチャは命令デコーダが強化されているのが特徴の一つである．正確にいえば，ドキュメントによっては，命令デコーダは QuantiSpeed の特徴には入っていない．しかし，Athlon で強化された命令デコーダは，Hammer でもそのまま受け継がれているようなので，ここでは QuantiSpeed に含めておく．

インテルの P6（Pentium II /Pentium III）アーキテクチャまでは，デコーダは対称的ではなかった（NetBurst では命令トレースキャッシュを使用するので事情が異なる）．しかし，Quanti Speed では対称的なデコーダを備える．これは，命令をデコードする効率に係わる．

P6 と QuantiSpeed は，どちらも，複数の命令を一度にデコー

〔図26〕
Hammerの分岐予測

ドするために，複数の命令デコーダを用意している．P6では二つの単純デコーダと一つの複雑デコーダで役割を分担している．一方，QuantiSpeedでは，同じ機能の命令デコーダ(ある程度複雑な命令をデコードできる)が3組対称に並べている．

命令の分類としては，単純な命令，ある程度複雑な命令，非常に複雑な命令に分類できる．命令の整列がうまくいっており，それらが単純な命令であれば，1クロックで3命令をデコードできるのは同等である．しかし，複雑な命令が混じる場合は事情が異なる．

P6では，単純デコーダで処理できない命令は複雑デコーダに渡される．複雑デコーダでもデコードできない命令はマイクロコード命令シーケンサでデコードされる．命令デコードはインオーダに行われるので，複雑な命令が混じる場合は，1クロック間で，1命令あるいは2命令ずつしかデコードできず，スループットが低下する．それゆえ，P6では，レジスタ-レジスタ間演算といった単純な命令を使用しないと性能が出ないといわれている．それに対しAthlonでは，1世代前のPentiumやK6アーキテクチャに最適化したコードでも，それなりの性能が出るとされている．

QuantiSpeedでは，単純な命令とある程度複雑な命令を処理する直接径路(Direct Path)と複雑な命令を処理するベクタ径路(Vector Path)に分かれて並列にデコードを行う．これは，AthlonではSCANステージ，Hammerではピックステージで選択される．

直接径路では，x86命令をデコードして，1クロック間に，三つのMacroOPを出力する．このMacroOPは，後のアーリデコードあるいはパックステージで結合され，RISCライクなμOPに変換される．ベクタ径路ではマイクロコードROMがアクセスされ，1クロック間に，最大3命令のMacroOPを出力する．これらが，専用デコーダでμOPに変換される．

上述のように，QuantiSpeedの命令デコーダでは，見かけ上，直接径路とベクタ径路の違いをなくしている．これは，複雑な命令が混じっていても，命令デコードのスループットが低下しないことを意味する．

● 分岐予測

投機実行が当たり前になっている最近のMPUでは，性能向上のために，分岐予測機能はとくに重要である．AthlonとHammerの分岐予測機構は，Athlonに分岐ターゲットアドレス計算器がない点を除けば，分岐予測テーブルのエントリ数の違いはあるものの，基本的には同じである．ここでは，Hammerの分岐予測について説明する．

Hammerの分岐予測は，**図26**に示すように，3種類のテーブル(キャッシュ)とターゲットアドレス計算器から構成される．

- 大域履歴カウンタ(Global History Counter)
  分岐の方向(分岐/不分岐)を記憶．
  2ビット×8Kエントリ(Athlonの4倍)
- 分岐ターゲットアドレス配列(Branch Target Address Array)
  2Kエントリ　(Athlonと同じ？)
- リターンアドレススタック(RAS：Return Address Stack)
  12エントリ　(Athlonと同じ)
- 分岐ターゲットアドレス計算器(BTAC：Branch Target Address Calculator)
  1個　(Athlonにはない)

〔図27〕Hammerのパイプライン

ステージ	分類	詳細	説明
1	フェッチ	フェッチ1	命令をL1キャッシュよりフェッチ
2		フェッチ2	
3		ピック	命令の整列
4		デコード1	x86命令をデコード
5		デコード2	中間コードに変換？
6		パック	個々の命令をパック
7		パック/デコード	命令をμOPに変換
8	実行	ディスパッチ	μOPをスケジューラ発行
9		スケジュール	スケジューラに格納
10		AGU/ALU	命令実行/アドレス生成
11		データキャッシュ1	データキャッシュアクセス
12		データキャッシュ2	
13	L2キャッシュ	L2リクエスト	
14		L2タグへのアドレス	Address to North Bridge
15		L2タグ	Clock Boundary
16		L2タグ，L2データ	SRQ Load
17		L2データ	GART/AddrMap CAM
18		L2からのデータ	GART/AddrMap RAM
19		データキャッシュへのMUX	Cross Bar
20		L1へライト，データ供給	Coherence/Order Check
21	DRAM		MCT Schedule
22			DRAM Cmd Q Load
23			DRAM Page Status Check
24			DRAM Cmd Q Schedule
25			Request to DRAM Pins
26			DRAM Access
27			Pins to MCT
28			Through North Bridge
29			Clock Boundary
30			Across CPU
31			ECC and MUX
32			Write Data Cache

フェッチした命令が分岐命令であるか否かは，L1キャッシュ格納時にプリデコードされて格納されている分岐選択情報(Branch Selector)によって判別される．そして，分岐命令をフェッチすると，大域履歴カウンタ(GHC)と分岐ターゲットアドレス配列(BTAA)を参照し，前者から分岐方向を，後者から分岐先アドレスを得る．分岐先アドレスが判明すれば，そこから投機的に実行を始める．

なお，分岐選択情報はL2キャッシュにも存在する．これはL1キャッシュからL2キャッシュへ追い出す場合に，L1キャッシュの情報をECC領域に書き込むようだ．さすがに，L2キャッシュへのリフィル時にプリデコードを行うのは回路が複雑になり過ぎる．

ただ，命令キャッシュでL1キャッシュからL2キャッシュへの追い出しが存在するかという疑問もあるが，2001年のMicroprocessor Forumでは，そのように解説されていたらしい．**図26**においても，Evicted Data(立ち退かされたデータ)という表現がある．ということは，L1キャッシュとL2キャッシュは排他的(Exclusive Cahce)になっていて，最新の情報をL1キャッシュに置くように，L1キャッシュとL2キャッシュで適宜エントリの置き換えが行われていることになる．これは，L1キャッシュのほうがL2キャッシュより短時間でアクセスできるためであろう．たしかに，Hammerのブロック図(**図25**)ではL1命令キャッシュとL2キャッシュの径路は双方向になっている．このあたりはAMDからの詳細な資料を待ちたい．

GHCはBTAAより多くのエントリを持つので，分岐命令の処理に，次のような場合が考えられる．

- GHCにヒット，BTAAにヒット
  1クロックのペナルティ
- GHCにヒット，BTAAにミス→BTACを利用して分岐アドレスを計算
  4クロックのペナルティ(BTACの時間は2クロック)
- GHCにミス→分岐するか否かはパイプラインの実行ステージまで不明
  11クロックのペナルティ(実行ステージまでの時間は9クロック)

ここでいうペナルティとは，逐次的な命令フェッチと比べて，分岐先をフェッチするまでに要するむだ時間のことである．これに1クロック加算したものが分岐命令の実行クロックといえる．HammerではBTACを追加したことにより，Athlonに比べて，GHCミス時のペナルティを大幅に減少させている．

● パイプライン

Hammerのパイプラインを**図27**に示す．AMDの説明によると，Hammerのパイプラインは，整数演算が12ステージ，浮動小数点演算が17ステージという．**図27**は整数パイプラインで，浮動小数点パイプラインは明らかにされていない．Hammerでは，Athlonでは1ステージだったL1キャッシュアクセスが，2ステージに分割されているのが特徴的である．デコードも2ステージかけて余裕をもたせている．明らかに，高い動作周波数をねらった設計である．それなのに，2ステージしか違わないというのは，1クロックに処理する論理を見直したためであろう．

まとめ

以上，実際のプロセッサのスーパースカラ構造について解説してきた．

たとえば，SH-4のパイプラインに関しては資料がほとんど存在しない．しかし，x86系に関しては資料は豊富である．そういう意味では，SH-4は筆者の想像，x86系は既存の資料の受け売りが多いことは否定しない．多少の誤りはご容赦願いたい．

文章の量はかなり多くなったが，内容的にはそれほど難しいことはいっていない．最近のMPUはアウトオブオーダなスーパースカラが常識のようになってきているので，スーパースカラの基礎をおさえておくことは必須であろう．

なかもり・あきら　フリーライタ

Appendix 2

携帯機器ではとくに重要な

低消費電力技術の原理

中森　章

はじめに

「消費電力＝性能」と考えられていたのは昔の話である．現在ではEWSでさえも低消費電力を考慮している．つまり，低消費電力で高性能という要求が強い．

低消費電力の利点は，システム構成の簡略化にある．電力が大きいと熱を発生する．すると熱により装置が誤動作するのを防ぐため，冷却機構の考慮が必要になる．あるいは電源の問題もある．消費電力が大きいと，それに見合う電源供給が必要になるので，強力な電源装置が必要になる．当然，システムの規模も大きく価格も高くなる．

これはPCの世界でも同様である．マザーボードから冷却ファンを取り除きたいというのが，第1の目的である．システム簡略化の次の目的は，携帯性の向上である．ノートPCでは電池寿命の長さが肝である．そのためには，できるだけ消費電力が少ないことが必須となる．このためには，MPUだけでなく，周辺機器も低消費電力化する必要がある．MPUはその第一歩である．

これらの要求を満たすため，1999年頃から低消費電力モードの採用がPC用MPUの売りの一つになってきている．たとえば，CrusoeのLongRun，AMDのPowerNow!やインテルのモバイルPentium IIIのSpeedStepなどである．AMDの発表ではSpeedStepは7W程度，PowerNow!は3W程度の電力が節約できるという(2000年当時)．これでもけっこう画期的な値で，電力は従来の1/10以下になり，それにより電池寿命が10～20％(SpeedStepの場合)，あるいは30％(PowerNow!の場合)長くなる．しかし，性能低下も著しく，低消費電力モードは無効化して使用する可能性が高いというのが，もっぱらの予測だった．

現在，低消費電力技術をとくに必要としているのは携帯電話やPDAの分野である．この分野では電池駆動が常識であり，1～15日程度の電池寿命を実現するために，10mW～200mWの超低消費電力が要求される．この要求を満たすのは，もはやPC用のMPUでは不可能であり，ARM，MIPS，PowerPCといった新しいアーキテクチャをもつMPUの存在する意義がある．

ここでは，低消費電力を実現するための基本原理について説明する．

● 低消費電力の基本原理

従来，いろいろな低消費電力技術が考案されてきているが，その基本原理はただ二つである．駆動電圧を下げることと動作周波数を下げることである．

電力は駆動電圧と消費電流の積で計算される．つまり感覚的には，

(電力)＝(電圧)×(電流)
　＝(電圧)×(電圧)/(抵抗値)　……オームの法則

であり，電圧の2乗に比例する．電力を下げるために駆動電圧を下げることは有効である．電圧が1/2になれば電力は1/4になる．

半導体の製造技術の向上につれて微細化が進み，低電力によるトランジスタ駆動が可能になってきている．たとえば，製造プロセスが0.25μmで2.5V，0.18μmで1.8V，0.13μmで1.5V，0.10μmで1.0Vである．つまり，製造プロセスを進化させることで，自然と電力も下がっていく．

また，電力は動作クロックの周波数(動作周波数)に比例することがわかっている．最近のMPUを構成するトランジスタはCMOS回路である．CMOS回路は，スイッチング時の状態遷移時にのみ電流が流れる．つまり，出力が変化するときのみ電流が流れる(**図A**)．これは状態をできるだけ変更しなければ電力が低減できることを意味する．

ところで，電気信号は動作クロックに同期して切り替わるので，動作周波数が高いほど電流値が大きくなる．つまり，動作周波数を下げることで，動作電流を下げることができ，結果として電力を下げることができる．

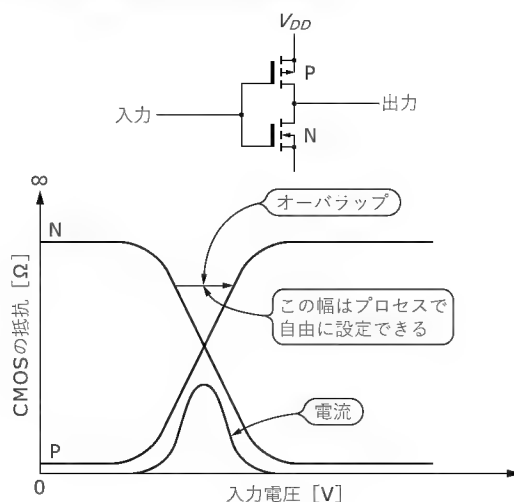
以上をまとめていうと，CMOS回路では一般に，

(電力)＝Σ{(寄生容量)×(電圧)×(電圧)×(周波数)}

という関係が成立することが知られている．これを示すCV^2fという表現はよく使われるので覚えておこう．

LongRun，SpeedStep，PowerNow!はどれも駆動電圧と動作周波数を動的に制御する技術であり，どのようなタイミングで制御するのかが，実現方法のキーポイントとなっている．

たとえば，CrusoeのLongRunは，電圧と周波数の組み合わせを複数用意し，MPUの負荷によって，それらを動的に変更するようになっている．負荷が軽い場合，まず，周波数を低下させ，その後，電圧を低下させる．ここでいちばん工夫を要するのが負荷の状態を判断する方法である．これは，アプリケーションプログラムが発行するコードモーフィング要求の頻度を監視することで実現していると思われる．しかし，その詳細な手法は公開されていない．

〔図A〕CMOSのNOT回路の動作

ここで，消費電力に関して興味深い報告を一つ．一般にMPUの動作周波数を上げると消費電力も増大する．しかし，実際のシステムにおいては必ずしも正しくない．動作周波数が速いと周辺回路が動作する時間が短くなるため，結果として消費電力が少なくなる場合もある．ただし，これは周辺機器が同時動作している通常動作時の消費電力の話である．長い期間の電池寿命に効いてくるのは，主として，スタンバイやサスペンドなど待機時の消費電力である．

● 動作電圧の動的制御

この方法は単純である．供給する電源電圧を上げ下げするだけである．当然，レギュレータなどの外部回路の補助が必要である．また，MPUが負荷状況を外部に通知する手段が必要である．これが，電圧を上げ下げするタイミングとなる．

インテルのモバイルPentium IIIで採用されたSpeedStepは複雑な制御をしない．AC電源をはずしたときのみ，自動的にクロックと電圧を2段階に制御する．650MHzまたは600MHz時は1.6Vで動作し，500MHz時は1.35Vで動作することで，消費電力を削減する．2001年7月に公開されたEnhanced SpeedStepでは，ソフトウェアで明示的にモード切り替えを行うことも可能である．これはモバイルPentium4-Mで採用された．

トランスメタのCrusoe（TM5400）では，電力管理用に5本の制御信号があり，32段階の電力制御が可能である．これは，主として，電圧制御用である．電圧は1.1V～1.6Vの間で0.05V刻みに変化できる．一方，動作周波数は自動的に変動する．200MHz～700MHzの間を33MHz刻みで変化する．MPUの負荷の変動は0.5μsごとに検出可能であるという．電圧を変更するのに要する時間は20μs以内で，アプリケーションの実行に影響を与えることはないといわれている．

AMDのK-6+，K6-III+，モバイルAthlon4，モバイルDuronなどに採用されているPowerNow!も同様の技術である．そのオートマチックモードでは，MPUの負荷を自動的に判断して電力制御を行う．その具体的な手法に関しては公表されていない．動作電圧は，2.0Vから1.4Vの間を数段階に分けて変更する．動作周波数の変更は，バスクロック（FSB）に対する倍率を変更することで実現する．PowerNow!は，LongRunと比較すると，動作周波数，動作電圧ともに変化の刻み幅が大きい．この意味で，PowerNow!は，SpeedStepとLongRunの中間的な技術といえる．

従来は，電圧の切り替えは2段階で十分としていたインテルも，Pentium-M（Banias）ではGeyserville-III（ガイザービルIII）という技術で多段階の電圧切り替えをサポートするようになった．0.85V時では600MHz動作，1.35V時は1.6GHz動作が可能である．この間で電圧と動作周波数の組み合わせを指定できる．現段階では1GHzという中間値が示唆されている．また，SpeedStep（Geyserville-II）とは異なり，Geyserville-IIIではCPU自体が負荷状況をモニタして，自動的に周波数と電圧の組を切り替えるようだ．つまり，ソフトウェアの変更をしなくても，自動的に省電力制御を行うことができる．

結局，LongRun，PowerNow!，Geyservilleは同じようなシステム仕様に近づきつつある．

● 動作クロックの制御

最近は，ほとんどすべてのMPUがPLL（Phase Locked Loop）を内蔵し，PLLからクロックが供給される．PLLは，原発振（FSBクロックなど）を逓倍することで，安定した高い動作クロックを生成する．動作クロックの制御は，PLLの逓倍率を変更すること，または，PLLの出力を分周することで実現できる．しかし，この手法はモトローラ社が特許を取っており，特許侵害を避けるため他社製MPUでの実現方法は意図的に不明確になっていることが多い．

（1）静的制御

これは，携帯電話の待ち受け時など，MPUが高速で動作する必要がないことが分かっている場合，ソフトウェアによって明示的に動作クロックを分周して低下させる方法である．専用命令の実行，あるいは，レジスタ設定によってクロックの分周比を変更する．

（2）動的制御

これは，ハードウェアによってMPUの負荷状況を監視し，動作クロックを変動させる方法である．トランスメタのLongRunやAMDのPowerNow!で実現されている．

MPU内のハードウェアによる監視を行わなくても，専用の入力端子を用意して実現する方法も考えられる．もっとも，これは監視回路をMPU外部に追い出したのと同じことであるが．

● クロック供給の制御

通常，MPUはいろいろなユニットの集合体で実現されている．そこで，MPUの動作に必要なユニットにのみクロックを供給する方法がある．

（1）静的制御

これは，低電力モードとして専用命令で提供されることが多い．低電力モードに移るとMPUは待機状態に入る．そして，割り込みが入ると待機状態から抜け出す．低電力モードには，クロックを供給する程度に応じて，スタンバイ，ウェイト，サスペンド，ハイバーネートなどと固有の名称が付けられている．たとえば，NECのV_R4131は**表A**（**a**）に示す五つの電力モードをもっている．これらの動作モードには，STANDBY，SUSPEND，HIBERNATEといった命令を実行することで移行する．また，**表A**（**b**）に示すように，ARM社のARM11でも同様の電力管理を実現している．

また，MPUによってはクロック制御ユニットをもつものもある．これは，各ユニットへのクロック供給を，ソフトウェアによって明示的に制御しようとするものである．MPUに備わっている機能ユニットでも，アプリケーションによっては，まったく使用しないものがある．これらのユニットにクロックを供給するのはむだなので，明示的に動作を止めようという考え方である．日立の携帯電話向けSH-Mobile（SH3-DSPをコアとする省電力プロセッサ群）では，自動的に（？），各周辺ユニットに供給するクロックを停止できるようだ．

（2）局所的な動的制御（ゲーテッドクロック）

低消費電力に関しては，電圧や周波数だけでなく，MPUのシステム設計の段階で，フリップフロップ単位に低電力のしくみをもたせる方法もある．これは，マスククロックまたはゲーテッドクロック（あるいはクロックゲーティング）と呼ばれる手法で，必要な場合にしかフリップフロップにクロックを供給しない技術である．MPU内部では，クロックを0から1へ，1から0へと変化させることだけで電力を消費する．このクロックを，CMOS回路の状態遷移に必要な最小限の期間しか動作させないという技術である．

具体的には，多ビットのレジスタやバッファに対し，選択信号（リード信号やライト信号）が出力されているときのみ，クロックを供給する（**図B**）．1ビット程度のレジスタ（フリップフロップ）では，クロックマスク回路の規模のほうが大きくなるので，意味がない．

（3）大局的な動的制御（ゲーテッドCTS）

MPUの内部回路にクロックを供給する場合，場所によってクロック間の遅延がないようにすることは非常に重要である．そのために，

〔表A〕電力モード

モード	PLL	内蔵周辺ユニット				CPUコア
		リアルタイムクロック	割り込み制御ユニット	バス制御ユニット	その他	
フルスピード	ON	ON	ON	ON	選択可能*	ON
スタンバイ	ON	ON	ON	ON	選択可能*	OFF
サスペンド	ON	ON	ON	OFF	OFF	OFF
EXサスペンド	OFF	ON	ON	OFF	OFF	OFF
ハイバネート	OFF	ON	OFF	OFF	OFF	OFF

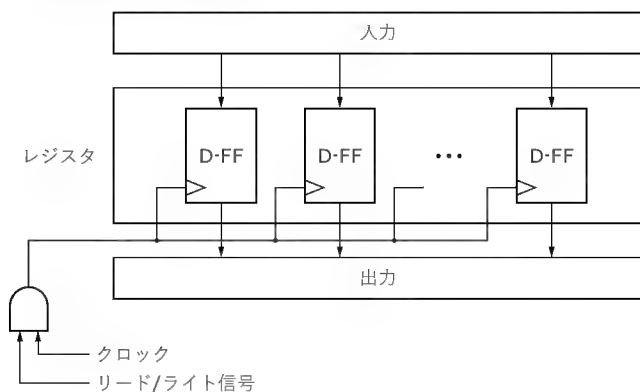
ON ：クロックを供給する
OFF ：クロックを停止する
＊ ：CMU（クロックマスクユニット）で選択

(a) V_R4131の電力モード

モード	コア	メモリ	電力	Runモードへの復帰
Run	クロックON 電力ON	クロックON 電力ON	アプリケーション依存	
Standby	クロックOFF 電力ON	クロックOFF 電力ON	リーク電流	割り込み デバッグ要求
Dormant	クロックOFF 電力OFF	クロックOFF 電力ON	メモリからのリーク電流	ソフトリセット
Shutdown	クロックOFF 電力OFF	クロックOFF 電力OFF	ほぼゼロ	リセット

(b) ARM11の電力モード

〔図B〕ゲーテッドクロック

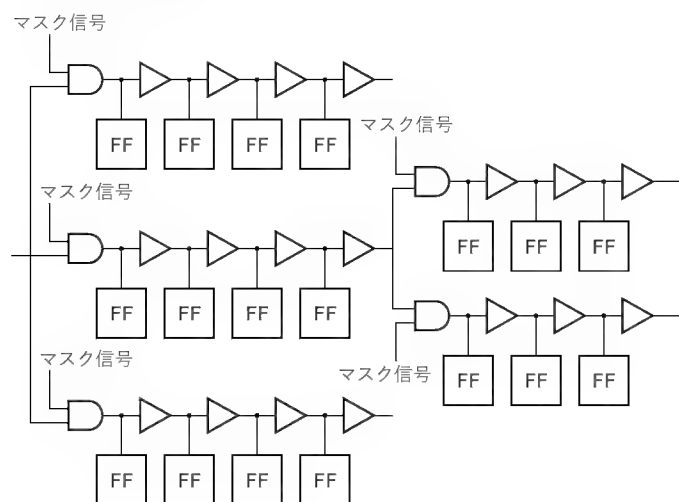
〔図C〕ゲーテッドCTS

類似した機能をもつ回路に対して共通のクロックツリーを設け，各クロックツリー内，異なるクロックツリー間で，その下にぶら下がるフリップフロップに見えるクロックの遅延が一様になるようにバッファ挿入を行う（**図C**）．これをCTS（Clock Tree Synthesis）という．

MPUが命令をデコードした時点で，その命令が駆動するクロックツリーを検出し，必要のないクロックツリーへのクロック供給を根元から止めてしまう．このようにして低消費電力を実現する方式をゲーテッドCTSという．この手法は，ある程度大域的にクロックを止めるので，通常のゲーテッドクロックよりも効果が大きい．一般的に，マスククロックとかゲーテッドクロックという場合は，このゲーテッドCTSのことを指す．

ゲーテッドCTSでは，クロック供給はハードウェアによって，自動的にクロック供給が行われる．しかし，大量のクロックマスク用ゲートでの遅延を揃えるために，高機能なツールが必要である．このため，最近まで，やりたくてもできない技術だった．大体，昔はクロックラインにロジックを挿入するなど非常識なことと考えられていた．

ゲーテッドCTSが叶わなかった昔，長い間使われない回路へのクロック供給を止めるという中間解が考案された．これは，外部回路でI/Oなどの状態を監視し，アイドル期間がある程度続くようなら割り込みを入力して，割り込みハンドラ内でソフトウェアによる電力制御（クロックを分周したり停止したりする）を行う手法である．

(4) その他の技術

MPUの内部回路はすべてが最高の周波数で動作する必要はない．クロックドメイン（一つのクロックが管理する領域）ごとに，処理の性質に応じて最適な周波数を選択することで，消費電力を最適化できる．たとえば，パイプラインを駆動するクロックは最高周波数であることが要求されるが，周辺回路に供給するクロックはそれ程高くなくてよい．とくに，割り込みのサンプリングは，かなり遅い周波数でも実用になる．

また，究極的な方法として，使用していないユニットの電源供給を停止する方法がある．これに関しては，対象となるユニットの再起動に時間が掛かるのが欠点である．MPU内の電流逆流防止などの考慮も必要である．

● キャッシュの電力制御

現在のRISCプロセッサはキャッシュの存在が必須である．しかし，MPUの内部でキャッシュがもっとも電力を消費する部分である．消費電力の半分がキャッシュによるものといっても過言ではない．このため，完全低消費電力を実現するためには，ゲーテッドクロックなどで，キャッシュ以外の部分の低消費電力化を行うだけでは不十分で，キャッシュそのものの電力を削減することが重要になる．

(1) ブロック分割

これは，キャッシュメモリを複数のバンク（ブロック）に分割して，必要なブロックのみを活性化してアクセスする方法である．具体的には，各メモリバンクは選択信号をもち，キャッシュをアクセスす

るアドレスの一部分をデコードすることにより，活性化するバンクを選択する．キャッシュの電力制御の方法として，ごく普通に行われている．

キャッシュメモリはSRAMで構成される．SRAMは，**図D**のようにメモリセルを2次元に配置し，アドレスを行(ロウ)アドレスと列(カラム)アドレスに分割して，アクセスするメモリセルの位置をデコードする．行選択で選ばれた複数のメモリセルが，ワード線を活性化することにより，記憶していたデータとその反転データをビット線に出力し，それをさらに列選択で選んだものをセンスアンプで増幅し，読み出す．

ブロック分割は，行選択をさらに細かく選択して行うことである．つまり，一部のワード線しか活性化しない．メモリセルはフリップフロップに入出力用のゲートを付加したものであり，このゲートはワード線を活性化して駆動する．駆動するメモリセルが少ないほど消費される電流が少ないのは明らかで，結果として低消費電力を実現できる．

(2) ウェイ予測

これは，nウェイセットアソシアティブキャッシュにおいて，ヒット/ミスの判別時にすべてのウェイの内容を同時に読み出すのではなく，予測した順番でウェイを読み出す方法である．キャッシュを同時に読み出さない分，電力が少なくなる．この場合，予測を正しく行わないと，キャッシュアクセスに時間がかかってしまうが，スーパースカラ構成を採る場合は，リザベーションステーション(早い話が命令キュー)でそのロス時間は給されてしまうといわれている．

ウェイ予測に関しては，基本特許が多く出ているので，安易に採用することは難しいと思われる．しかし，堂々とウェイ予測を謳うMPUが発表されている現状を考えると，抜け道はいくらでもあるのだろう．

(3) キャッシュの階層化

キャッシュは，通常，L1キャッシュ，L2キャッシュ，L3キャッシュと階層化して使用される．これは，キャッシュのアクセス時間と内蔵できるキャッシュサイズとの兼ね合いで分割されていることが多い．つまり，L1キャッシュ，L2キャッシュ，L3キャッシュの順にキャッシュサイズが増加していくのが普通である．そして，L1キャッシュはL2キャッシュの内容の一部を，L2キャッシュはL3キャッシュの内容の一部をキャッシングする(ビクティムキャッシュはそうなってないが)．

このような階層構造は，消費電力を下げる効果もある．一般的に，アクセスするキャッシュのサイズが大きいほど電力を消費するので，時間的，空間的に，できるだけ少ないサイズのキャッシュへのアクセスを優先することで低消費電力化を実現できる．この方式は，特殊な形態のブロック分割といえるかもしれない．

アドレス変換に使用するTLBでは，その内容をキャッシングしたマイクロTLBを最初にアクセスすることがあるが，これも同じ発想である．

● トランジスタレベルの低消費電力化

この分野は筆者の専門ではないが，簡単に説明する．

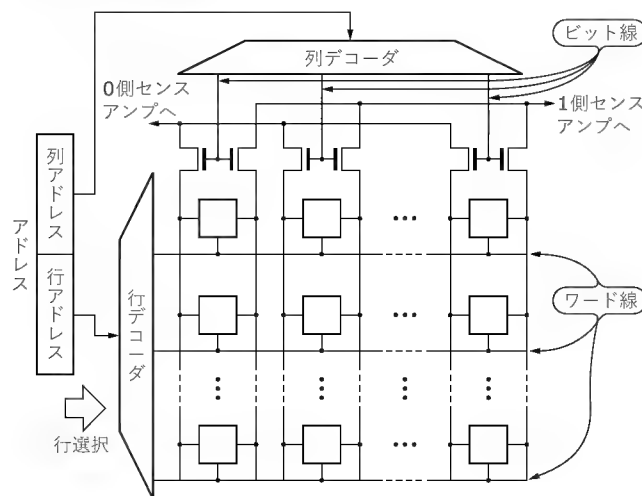
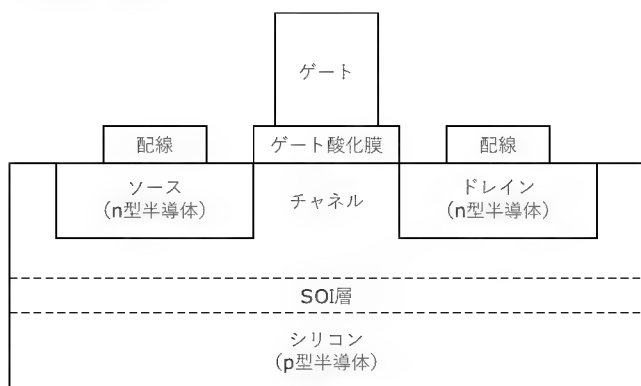
トランジスタの集積度はムーアの法則にしたがって増大してきた．1971年に登場した4004と最新のPentium4を比較すると，動作周波数は約2万倍に，トランジスタ数は約2万4000倍にまで増大した．しかし，動作電圧は低下しているが，動作周波数とトランジスタ数の増加とともに消費電力も増加している．この消費電力がMPUを発熱させる要因となっている．インテルの試算では，MPUの発熱は，2007年には核反応炉と同レベルになり，2010年を過ぎた頃にはロケットの噴射口に匹敵し，程なく太陽の表面温度に等しくなるという．その意味でも，トランジスタの低消費電力化は必須である．

(1) リーク電流

半導体製造プロセスの微細化が進むと，当然，トランジスタの大きさは小さくなる．トランジスタとは，シリコンの上に，シリコンと逆極性のソース領域とドレイン領域を形成し，それをゲートで繋ぐ(**図E**)．ゲートに電圧をかけるとソースとドレインの間に電流が流れ，これがスイッチのように働く．

しかし，微細プロセスでは，ゲート長が短くなり，トランジスタが導通し始める電圧(しきい値：V_{th})が低下して，ゲートに電圧をかけなくても漏れ電流(リーク電流)が流れてしまう．このリーク電流(オフリーク電流，または，チャネルリーク電流)のせいで，トランジスタが動いてなくても，電力を消費してしまう．また，これでは，トランジスタがONなのかOFFなのか判別できないので，正常動作のためには，リーク電流を超える電流を流す必要がある．これによっても消費電力が増大する原因となる．

あるいは，ゲート長の縮小化にともない，ゲート酸化膜(絶縁膜)もほぼ比例して薄くする必要がある．しかし，膜厚が薄くなること

〔図D〕SRAMの構成

〔図E〕nMOSトランジスタ

により，トンネル効果で，絶縁膜を通して流れるリーク電流（ゲートリーク電流）が無視できなくなる．

オフリーク電流とゲートリーク電流は，消費電力を増大させる要因になる．これは，電池駆動の機器においてはとくに深刻な問題であり，リーク電流を削減するための手法がいろいろ考案されている．

リーク電流を低減させる抜本的な方法を以下に示すが，とりあえず行われるのが，微細プロセスになるほど動作電圧を下げるという方法である．トランジスタを駆動する電圧が低ければ，流れる電流も小さく，消費電力が下がるという理論である（消費電力は電圧の2乗に比例する）．

（2）オフリーク電流対策

オフリーク電流対策としては，ソースやドレインを形成するシリコン層の厚さを薄くする方法がある．シリコン層にソースやドレインを形成すると，その影響による空乏層（depression region）[注]の広がりがシリコンの厚みに対して小さく，空乏化されてないシリコンがゲートの下に残る．このシリコンがフローティングボディ（どこにも接続されていない電極）となり，その電位がトランジスタ特性に影響を与える．シリコン層を薄くしてゲートの下のシリコンを完全に空乏化するとフローティングボディがなくなり，また電流が通過できる範囲が狭まる（抵抗が増える）ため，オフリーク電流を低減できる．しかし，シリコン層の厚みはV_{th}に影響を与えるので，これを薄くするとV_{th}が下げられる反面，トランジスタの制御が難しくなる．したがって，闇雲にシリコン層を薄くすればいいというものでもない．

インテルは完全空乏型のシリコン層を使用することで，部分空乏型よりもオフリーク電流を1/100にできると発表している．このトランジスタは完全空乏型基板トランジスタ（Depleted Substrate Transistor），略してDSTと呼ばれる．DSTでは，V_{th}を下げられるので，トランジスタの動作電圧を低減したり，トランジスタのON/OFF動作 を高速化したりするのに役立つ．DSTでは，部分空乏型 SOIトランジスタとは異なり，フローティングボディ効果が発生しなので，従来のトランジスタと同じ方法で回路設計が行えるのが利点であるとして，IBMを牽制している．

（3）ゲートリーク電流対策

ゲートリーク電流対策としては，ゲート酸化膜の素材を変更することが考えられている．ゲート酸化膜とは，ソースとドレイン間を流れる電流をゲートに流れ込まないように絶縁するための絶縁膜で，この部分が薄ければ薄いほどトランジスタは高速に動作する．しかし，ゲート酸化膜が薄くなるとゲートリーク電流が増加する．そこで，ゲート酸化膜を厚くする方法が考えられるが，できることならある程度の薄さも維持したい．

これを解決するため，ゲート酸化膜に誘電率の高い（high-k）材料を用いることで，誘電率の比率で酸化膜を厚くすることが可能になり，リーク電流が減る．これは，高速化技術のために提案されているlow-k材料および銅配線とは対極的であるが，こちらは配線に関する技術である．配線に関しては，寄生容量が電流の流れを妨げ，電力を消費する．このために，層間膜は誘電率の低い材質（low-k）や電気的抵抗の少ない銅配線が好まれるのである．

ゲート酸化膜に何が最適な素材であるかは，半導体メーカーが力を入れて研究している分野であり，現状，これといった決定版はない．現在，ゲート酸化膜には二酸化シリコン（SiO_2）を使うことが多い．high-kの絶縁膜の目的は，物理的には，厚い膜で薄いSiO_2と同じ電気特性（ゲート容量）を実現することでる．しかし，将来的には，インテルは誘電率が二酸化シリコンの約5倍の酸化ジルコニウム（ジルコニア：ZrO_2），酸化ハフニウム（HfO_2），二酸化チタン（TiO_2），五酸化タンタル（Ta_2O_5）などを使用することを提案している．AMDは誘電率が約2倍の窒化シリコン（Si_3N_4）を使用することを提案している．これらを使用することで，二酸化シリコンと同じ電気的性能を発揮しながら，ゲートリーク電流をその1/10,000～1/1,000に抑えることができるといわれている．

ただ，high-kの絶縁膜は，二酸化シリコンとは違って製造が難しく，さらに，シリコンとの相性が悪く信頼性に問題があると言われている．また，high-kの絶縁膜では，SiO_2に較べるとキャリア（電子や正孔）の移動度が落ちる（つまり，動作速度が落ちる）という問題もある．これらを解決するのが今後の課題である．新しいゲート絶縁膜が主流になる時期は，早くて，ゲート長が65nmの世代と言われている．

2002年12月に開催されたIEDM（International Electron Device Meeting）では，high-kの絶縁膜としてHfO_2が一般的になった感がある．これの膜質の改良や製造性の改善のためにNやAl，Siを添加する論文が多数提出されている．

（4）SOI技術

あるいは，SOI（Sillicon On Insulator：絶縁体の上のシリコン）という技術で，ソース-ドレインとシリコン基板の間に二酸化シリコンによる極薄の絶縁層を組み込む方法がある．SOI層を加えることで，シリコン基板とソースあるいはドレインの間の寄生容量を低減することにより，ソース-ドレイン間の抵抗を減少させ，電流の流れを20～30％よくする技術である．同じ性能（電流の量）であれば，電源電圧を下げることにより消費電力をほぼ半減できる．まあ，これは，低消費電力というよりも高速化技術である．

SOIという技術は，30年以上前から半導体メーカーが研究をしている．その中で有名なのは，ハリスセミコンダクター社のSOS（シリコンオンサファイヤ）である．しかし，これは非常に高価なため実用にならなかった．現在は，二酸化シリコンを使用して比較的安価に作られているが，今後も新たな材質の発見が望まれる．

まとめ

今後，ますます重要な技術になると考えられる低消費電力技術について述べてきた．その根本原理は，低い動作周波数と低い動作電圧を実現すること，回路の内部状態をできるだけ変化させないこと，一度に駆動する回路を減らすことである．これらを組み合わせることで，今後も，いろいろな制御方式が考案されていくことと思われる．トランジスタレベルでの低消費電力化については，現在発展途上というところだろうか．

参考文献

1）中川靖，「64ビットRISCマイクロプロセッサ　VR4131」，『NEC Device Technology』，2001，No.74

なかもり・あきら　フリーライタ

注：電荷を運ぶキャリア（電子または正孔）が存在していない領域のこと．空乏があると電流の流れが悪くなる．

小型・著作権保護機能を特徴とする

SDIOカード開発入門

第1回 SDIOカードの現状

井手野雅明

はじめに

現在，SDIO(Secure Digital Input/Output)関連の情報や資料は非常に少なく，SDIOカード(カード側)はもちろん，SDIO対応のモバイル機器(ホスト側)を開発したい場合も，いったいどこから手をつけてよいのかわからない場合が多いだろう．そこで全6回の予定で，SDIOカードについて詳しく解説する．これからSDIOの設計開発を検討される方の道しるべとなれば幸いである．

連載の予定を簡単に示すと，次のようになる．

- 第1回　SDIOカードに関するさまざまな市場の状況や将来について
- 第2回　SDIO標準規格について
- 第3回　IEEE802.11b対応のSDIO無線LANカードについて
- 第4回　SDIOカードを設計するための具体事例(前編)
- 第5回　SDIOカードを設計するための具体事例(後編)
- 第6回　今後の市場展開などまとめ

開発事例などでは，実際にPalmなどのPDAを使ってSDIOカードを動作させるところまでを解説するなど，できるだけ具体的な事例を紹介していく予定である．

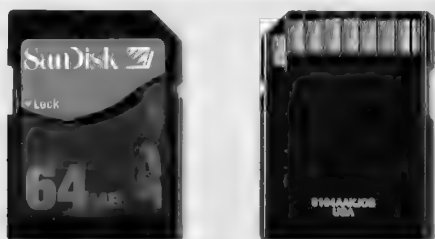
高集積サブシステムSDIOカードの登場

SDIO(エスディーアイオー)は，SDアソシエーションで策定されたSDメモリカードと同様の規格の一つである．一般的によく知られるSDメモリカードと比べると，その外形(長さ32mm×幅24mm×厚さ2.1mm)の長さ方向に関する規格だけが異なり，横幅並びに厚さに関してはSDメモリカードとまったく同じ形状をしている．端子数もSDメモリカードと同じ9ピンで配置位置も変わらず，電気的条件も同等である(**写真1**)．

SDIOカードの普及が待たれる理由の一つとしては，世界的に広がる機器のモバイル化または小型化の流れにある．ユーザーはいろいろな機能を一つの機器で実現し，しかも長時間ストレスなく使用できるサイズを求める傾向が見られる．しかし，これらすべての要求にこたえようとすると“機能増加→コスト増加”となり，さらに機器サイズの肥大化，消費電力の増大など，ユーザー要求にこたえようとすればするほどユーザーの満足を得ることが難しくなるという矛盾をはらんでいる．

SDIOは，このような多様化するユーザー要求をユーザーが個別にカードを選択することにより，機器本体はソケットという非常に安価でかつ構成しやすいソリューションを機器に搭載するだけで，要求機能の多くを実現できるようにしたものだ．一方，カード開発側にとってもSDアソシエーションに加入(機密保持書と同意書にサインが必要)すれば，SDIOカードの設計開発および販売を行うことができる．

SDIOカードは小型・薄型でありながら，PC(PCMCIA)カード(以降PCカード)，コンパクトフラッシュカード(以降CFカード)と同等以上の機能を提供する，世界最小のカードソリューションであることから(**写真2**)，これからも多くのSDIOカードが世に登場すると期待されている．

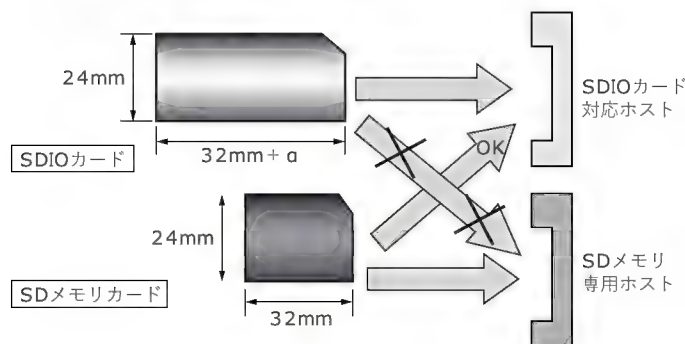
〔写真1〕SDメモリカードとSDIOカード

(a) SDメモリカード(表面と裏面)

(b) SDIOカード

SDアソシエーション

SDアソシエーションは，1999年8月に松下電器産業と東芝，SanDisk Corporationの3社で設立された，おもにSDメモリカードの共同開発，業界標準化およびプロモーションを目的とした団体である．現在会員数は588社(2003年5月現在)に達する規格標準団体となっている．SDアソシエーションではGeneralメンバとExecutiveメンバの2通りのメンバカテゴリがあり，それぞれ年会費が異なる．実質的な違いは，Generalメンバが制定された規格を入手し使用する権利を有するが，Executiveメンバは実際の規格制定にも参画できる権利を有する．

基本的にSDメモリカード規格，SDIOカード規格は最低でもGeneralメンバにならないと入手ができないしくみになっており，SDアソシエーションのWebページからダウンロードできる申込書(Membership Application)に必要事項を記入してメールもしくはFAXすることにより，後日SDアソシエーションから加盟承認の連絡がある．この機密保持契約書に署名捺印し年会費を支払うことによってはじめて加盟が成立する．詳しくはSDアソシエーションのWebページを参照してほしい(http://www.sdcard.org/)．

したがって，SDメモリカード標準規格とSDIOカード標準規格は，同じSDアソシエーションという親から生まれた双子のような関係だといえる．しかし，いくつか注意点がある．基本的にSDメモリカードのみに対応しているホスト機器(たとえば，現状のデジタルカメラなど)には，SDIOカードを挿入することはできるが動作はしない．SDIOカードを使用するにはSDIO標準規格に適合したホスト機器である必要がある．SDIOカードに適合したホスト機器は，SDメモリカードを使用することもできる(**図1**)．またSDアソシエーションでは，BluetoothやGPSなどのアプリケーションごとの規格も制定しており，これらはSDIOカード標準規格とあわせて各種アプリケーションに対応したSDIOカードを実現できるよう指針を与えている．

〔写真2〕各種無線LANカードの比較
(下：PCカード，左上：CFカード，右上：SDIOカード)

SDメモリカードの特徴

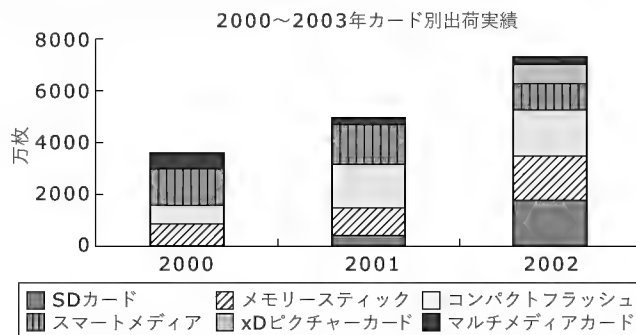
SDメモリカードの“SD”はSecure Digitalの略で，強力な著作権保護機能を有するメモリカードであることを表している．一般的には小さいカードというイメージが先行しているが，このSDメモリカードが誕生した背景には，音楽，映画，写真など著作権を有するコンテンツ配信において，データの違法コピーを防ぎ，e-Commerceを実現することが念頭に置かれている．

記憶容量は，現状では8Mバイト～512Mバイトまで各種容量のものが市販されおり，今後はより大容量の1Gバイト，さらには32Gバイトの容量をもつメモリカードも検討され始めている．また2003年3月には，SDメモリカードよりさらに小型のMini-SDメモリカードがアナウンスされたことからも推測できるように，携帯型情報AV端末機器への対応がおもなターゲットとなり始めている．

2002年の全世界製品出荷ベースでの実績では，SDメモリカードはメモリースティック，コンパクトフラッシュを抜き1,800万枚と世界第1位の出荷量に到達した(**図2**)．これらは搭載され

〔図1〕SDメモリとSDIOカードの関係

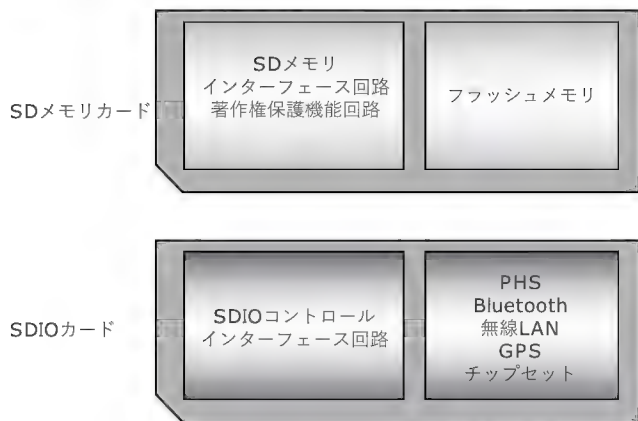
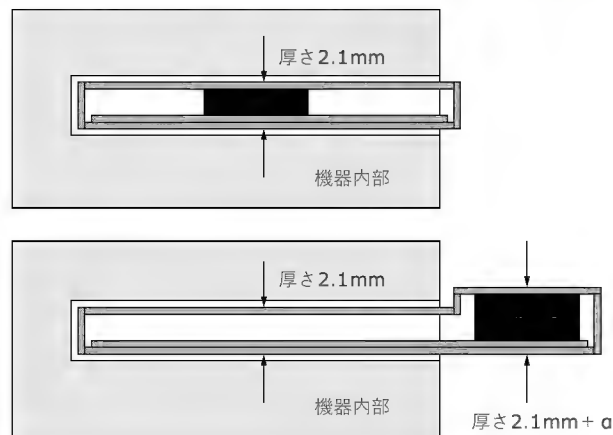
〔図2〕メモリカードの出荷実績

注：Techno Systems Research調べ

る機器，たとえば携帯電話やPDA，そしてデジタルカメラなどの小型機器に採用された影響が大きく，またSDアソシエーションに加盟している企業がすでに900種を超える多くのSDメモリカードに対応した製品を出荷していることにもよる．将来的にはこの強力な著作権保護機能を活かしたサービスの展開もあわせ，ますますSDメモリカードの需要が拡大するであろう．

SDメモリカードとSDIOカードの違い

SDIOカードは基本的にSDメモリカードと用途が大きく異なり，機能も異なっている．SDメモリカードは情報を蓄積または保護することがおもな機能となるが，SDIOカードはBluetoothや無線LAN，PHS，GPSなどのいわゆるPCカードやCFカードに見られるようなI/O機能を搭載しているカードである．つまり情報を蓄積するのではなく，処理するカードと考えたほうがわかりやすい．

SDメモリカードとSDIOカードの内部構成を**図3**に示す．SDメモリカードは，SDメモリインターフェース回路と著作権保護機能を有する回路にフラッシュメモリが接続されている．

〔図3〕SDメモリカードとSDIOカードの内部構成の違い

〔図4〕SDIOカードの実装上のサイズ

構造的には非常に簡単に示してあるが，実際はかなり複雑な処理を内部で行っており，確実な情報の蓄積と保管そして著作権保護を行っている．

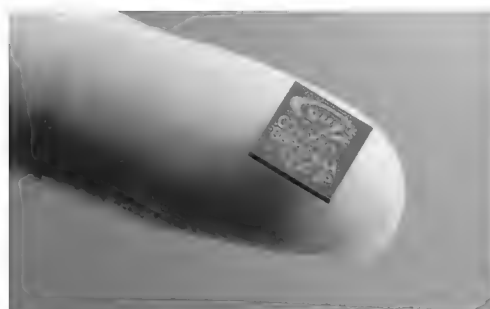
一方，SDIOカードを実現するには，まずSDIO標準規格Ver1.0（現在の最新版規格）に適合したSDIOコントローラインターフェース回路を用意する必要がある．これなしではSDIOホスト標準規格に対応したSDIOホストデバイス，または同等の機能を実現できるCPUとの通信を行うことができない．これらのインターフェース回路はIPとしてリリースされているが，一般的にはすでに市販されている個別専用LSIを使用するのが妥当である．本連載では，このSDIOコントローラインターフェース回路としてSD-Path CG100（シイガイズ，**写真3**）を使用して解説を進める予定である．

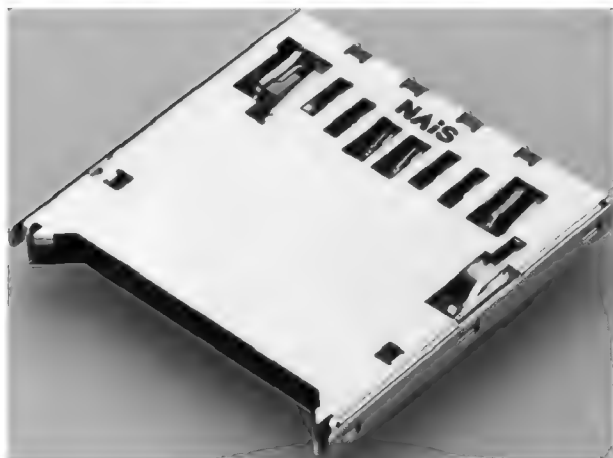
SDIOカードではSDIOコントローラの後に，PHSやBluetooth，無線LANなどのチップセットを接続して機能を実現することになるが，実装サイズが非常に小さいことから各種チップセットの厚さについて考慮する必要がある．チップセットの構成上，SDIO標準規格の厚さ2.1mmに収容できない場合は，SDIOカードを機器に挿入した際に機器外に飛び出すエリアに膨らみをもたせて実現する方法がある．しかし，この方法ではカードサイズが長く大きくなり，せっかくの小型というSDIOのメリットを打ち消してしまう（**図4**）．

基本的にSDIO標準規格では，長さ方向に関しての規定はないが，カード強度の問題がある以上，搭載する機能とチップセットの見きわめが重要となるので，CG100のように厚さ0.9mm程度の部品を選定するのが賢明といえる．ちなみに，SDメモリカードの規格では長さ方向にも規定があるため，カードサイズの拡張はできない．

SDIOカードが必要とされている背景

SDメモリカードはその小型な形状が評判となり，多くのモバイル機器で採用が相次いでいる．しかしながら市場の要求は単にメモリとしてのカードに加え，通信機能やカメラ，GPSなどの追加機能をカードで実現する要求が出てきた．既存のPC

〔写真3〕SDIOコントローラインターフェースIC　SD-Path CG100（シイガイズ）

〔写真4〕SDIOに最適なソケットの例(松下電工)

〔図5〕SDIOカードとSDIOホスト対応機器の増加

カードやCFカードでは，すでにこれらのような複数の機能をもつカードは存在しているものの，その外形的要因により小型モバイル機器での採用は見送られることがあった．そこに登場したのが，もっとも形状の小さいSDIOカードである．

また，SDIOカードのピン数や物理形状はSDメモリカードと同じサイズなので，基本的にはSDメモリカードとSDIOカードでは同じソケットが使える．現在，SDメモリカードおよびSDIOカード用のソケットは数社から発売されているが，その中から代表的なソケットを**写真4**に紹介する．このソケットは幅28.2mm×長さ28.4mm(端子含まず)×厚さ2.7mmのSDIO対応のソケットで，最大の特徴はソケットの表面/裏面ともに金属のシェルで覆われている点である．SDIOカードでは，ESDやEMI対策が求められており，表面と裏面が金属で覆われているこのソケットはうってつけといえる．

すでに説明したように，今のところSDメモリカード専用機器にSDIOカードを使用することはできないが，今後の展開としてはSDメモリカードとSDIOが兼用で使用できるSDIOホストデバイスを搭載した機器が多く登場する予定である．したがって，SDIOカードはその小型形状と高機能を兼ね備えたソリューションとして市場から求められることは必至であり，カード供給側としてはいろいろなアイデアを盛り込んだSDIOカードを実現し市場に供給することができる大きなチャンスとなる．

SDIOカードを取り巻く現状

SDIOは規格が制定されてからほぼ3年が経過している．実質的には2001年10月のCEATECで発表された東芝のBluetoothカードが最初のSDIOカードといわれている．その後，各社より開発着手または発売開始のプレスリリースが発表されている．

唯一の問題点といえば，SDIOホストを搭載した機器ラインナップの市場供給量といわれるが，少なくともPDAの世界ではPalm社および東芝がいち早くSDIO対応製品を出荷し，COMPAQも同様にSDIOホスト搭載のPDAをリリースしている．よって，現在ではPDA業界では，ほぼSDIOが標準となりつつある傾向にあるようだ．

一方ほかの機器では，PCでSDIOホスト対応機器が出始めたのを皮切りに，デジタルカメラやデジタルビデオカメラ，携帯電話などでSDIO採用の検討が始まって(**図5**)いる．さらに最近では，SDIOカードに対する要求として新たに低消費電力化が付け加えられた．

これらのことにより，過去から現在にかけてだいたい一巡した機器本体へのSDメモリカード専用ホストの搭載から，今後はSDIO対応ホストの搭載へ移行することになる．なぜならば，SDメモリカード用ソケットとSDIO用ソケットは同じ大きさ(同一ソケットで兼用)であるため，製品自体の筐体設計などを変えずに高機能なSDIOカードが使えるというメリットが出てくるためである．さらに付加価値をつけ販売拡大をねらうSDIO対応機器が登場する見込みである．

SDIOカードの将来性

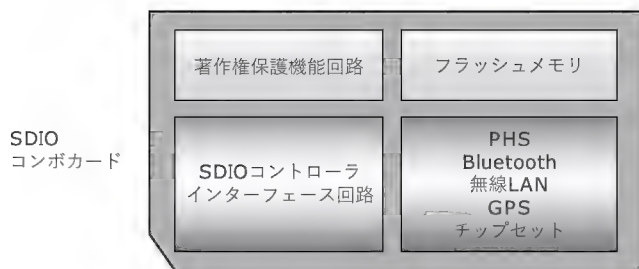
SDIOカードは現時点では単機能，つまりBluetooth単体や無線LAN単体というような製品ラインナップとなっているが，将来的にはこれら機能が複合化される可能性がある．現時点で注目を集めているのがBluetooth＋IEEE802.11b無線LANの複合カードや，IEEE802.11b＋11gのマルチ無線LANカード，無線LAN機能＋デジタルカメラ，PHS＋無線LANなどが挙げられている．また，最近新しく発表されたMini-SDでSDIOを実現しようという発想も出てきている．

そして現在市場から要求されているのが，SDメモリをSDIOカードに搭載することである．これはSDIOコンボカードと呼ばれている．SDIOコンボカードは**図6**のように，従来のSDメモリカードをそのままSDIOカードに取り込む方法で，各種ア

プリケーションを展開するに大きな武器となる．具体的にはSDIOデジタルカメラで撮影した高画質写真をそのまま内蔵のSDメモリに格納したり，SDIO-GPSカードでは地図情報を内蔵SDメモリに格納しておくことができる．ただし，SDIOコンボカードを実現するには別途契約が必要となるため，検討に際してはSDアソシエーションにコンタクトを取ることが最優先となるので注意してほしい．

SDIOカードのマーケット概要

2003年6月の段階ですでにいろいろなカードが発売されている．発売を予定しているカードを含めると，筆者が調べた範囲だけでも**図7**のようになる．これ以外にも多くのカードがリリースされていると思われ，相当数のSDIOカードが既に市場に出ていることになる．

比較的市場が大きいとされているのは無線系のカードであり，Bluetooth，IEEE802.11b無線LANカード，PHSカードなどが挙げられる．とくにBluetoothは近年チップセットが集積化され，かつ価格も値頃感が出始めたこともあり，多くのカードベンダがこの市場への参入機会を伺っている状況である．消費電力も少なくSDIOカードを実現するには非常に手ごろな仕様であり，今現在としてはホットな市場といえる．このBluetoothでのSDIOカード市場では価格競争が展開される可能性が大きいが，市場が大きいため複数のカードベンダが共存できる市場ともいえる．

次にホットなのは無線LANカード市場といえる．これもBluetoothと同様でカードという機能性を大きく活かせるアプリケーションであり，しかも昨今のホットスポットなどの登場でモバイル型機器への搭載が急務となり，各カードベンダが開発を始めている．この市場で優位に立つには，他のカードにはない独自性，たとえば低消費電力化などが必須となる．

その他の市場としてはデジタルカメラカード市場が比較的競争が激しいといえるかもしれない．デジタルカメラは，CMOSセンサなどの消費電力の少ない撮像素子を応用することにより，条件の一つである低消費電力化を実現している．モバイル型機器で要求されるのはむしろ写真ではなく，情報をクリップする用途，すなわちイメージの入力として使用されるケースが圧倒的に多い．よって，これらのカードでは機能的に接写能力&ディジタルズームといった両面での機能を充実させる必要があるといわれている．

最後にGPS，スキャナ，チューナなどのSDIOカード市場であるが，現在調べた限りでは存在はしていないようだ．GPSなどはこれから各種サービスと合体するかたちでの需要が期待されており，実際に設計開発を行う場合でも，すでにチップセットなどが存在していることから，実現化がいちばん近い存在ではないかと思われる．

今後登場が期待されるカードとしては，IEEE802.11g無線LANカード，地上デジタル波放送テレビチューナなどといわれているが，現状ではチップセットおよびカード集積と消費電力などの問題があり，なかなか前に進めないのが現状である．しかしながら，SDIO対応機器が今後増加するにともない，まったく新しい機能をもったSDIOカードが登場するのも時間の問題かもしれない．

〔図6〕SDIOコンボカードの内部構成

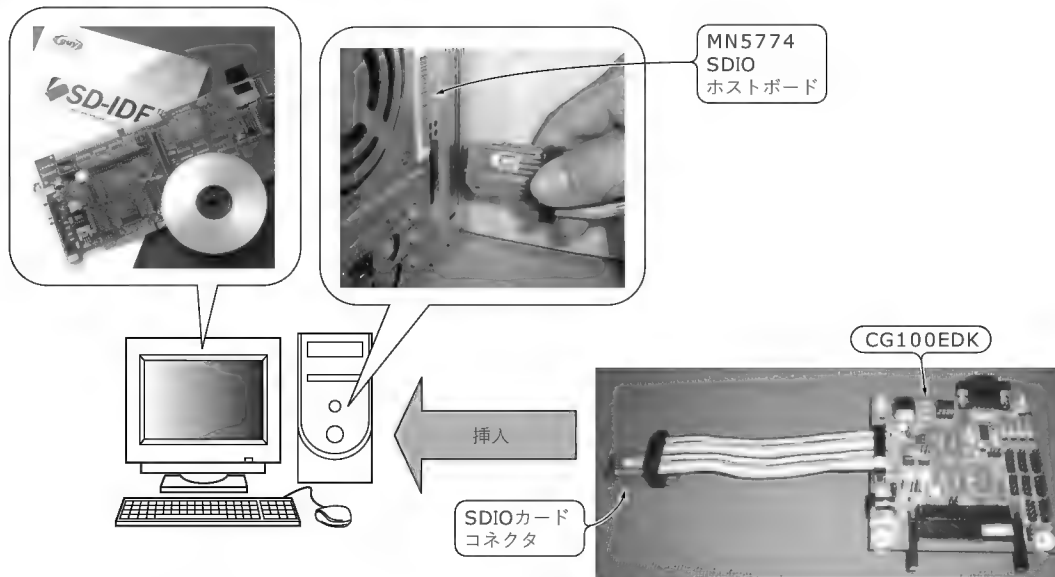
〔図7〕2003年6月現在のSDIOカード

SDIOカードを開発するには？

SDIOカードを開発する場合，まずはSDアソシエーションに加入することが先決である．加入後はSDIO標準規格書をメンバ専用ページからダウンロードし，内容を理解することが必要となる．しかし，初めてSDIO標準規格書を見てもなかなかすぐには理解することができず，実際に設計環境を構築して理解しながら開発を進める形をとるケースが多いようだ．

加えて開発を進める場合，実現しようとしているアプリケーションによって，SDアソシエーションで規定されているアプリケーション規格に適合させる必要が出てくる場合があるので，SDアソシエーションのメンバページをよく確認し，常に最新の資料をダウンロードしてSDIO標準規格とあわせて理解することが必要となる．

基本的にSDIOカードは，SDIOコントローラデバイスとアプリケーションチップセットの大きく分けて2部の構成になる．

〔図8〕CG100開発キット

注：SD-IDEはSD-IDEソフトウェアとリファレンスボードCG100EDKで構成されており，別売不可

したがって，対応するアプリケーションのチップセットのインターフェースと，SDIOコントローラデバイスのインターフェースが合うかどうかも事前に検討が必要である．

具体的なSDIOカードの開発方法は今後の連載で解説するとして，ここでは開発を始める前に用意しなければならない設計開発環境について，具体例を紹介する．

SDIOカードを開発するには，ホスト側環境とカード側環境を整備する必要がある．また，重要なソフトウェア，とくにドライバソフトウェアの開発をあわせて行えるようにしておくことが重要である．もし，OSプラットホームをWindows CE系にするのであればSDIO Now!(ビースクウエア)を購入することをお勧めする．

さて，実際の開発環境であるが，一から環境を構築するのは時間がかかるだけではなく，ミスによる設計開発品質の低下と手戻りによる時間的ロスが発生するためにあまりお勧めできない．本連載ではSDIO開発環境ツールのSD-IDE(シイガイズ)を使用してSDIO標準規格を理解しながら設計開発を進める．

SD-IDEは現在，OSプラットホームとしてWindows XPとWindows CEが用意されている．構成としては，SDIOソケットとSDIOホストデバイスであるMN5774が搭載されているSDIOホストボード(松下電器産業)をPCIバスでPCに接続しSDIOホスト側環境とする．カード側の環境は，SDIOカード形状のコネクタをもつCG100リファレンスボード“CG100EDK”(シイガイズ)を使用している．なおSD-IDEとCG100EDKは，CG100デバイス開発専用としてセットで構成しているので別売りは行っていない(**図8**)．またSDIOホストボードは，今のところMN5774のみの対応となっている．SD-IDEに関する詳細情報は次のURLを参照してほしい．

http://www.c-guys.jp/

まとめ

以上，SDメモリカードとSDIOカードの違いからSDIOカードが置かれている現状，そして将来性と実際に設計開発行ううえでのポイントなどを解説した．次回では，肝となるSDIO標準規格の概要をわかりやすく解説する予定なので，SDIOカードを開発するうえでの助けとなれば幸いである．

SDIOマーケットの拡大はより多くのSDIOカードの登場があって初めて成り立つものであることから，新しいアイデアを満載した，素晴らしいSDIOカードの登場を願ってやまない．

いでの・まさあき　シイガイズ(株)マーケティング部

やり直しのための信号数学

第18回

DCTとフィルタバンク

三谷政昭

前回は，DCTの物理的な意味付けと，一般式を示し，DCTによる信号解析の考え方や特徴を中心に，具体的数値例に基づいて解説した．

今回は，DCTとフィルタバンク（複数のディジタルフィルタを並列構成したもの）の関係について説明する．すなわち，DCTが周波数分解，IDCTが信号再合成の機能を有することの数式上の裏付けを中心に，具体例を示してわかりやすく述べる．なお，フィルタバンクの考え方には，オーディオや画像などの信号データの圧縮，ウェーブレット変換の実現および設計などの土台になるヒントが数多く含まれているので，じっくりと読み進めていってもらいたい．（筆者）

DFTと信号解析ディジタルフィルタ

最初に，Nサンプルのディジタル信号$\{x_n\}_{n=0}^{n=N-1}$に対するDFT計算が，信号解析ディジタルフィルタ（以後，信号解析DFと記す）として実現できることを思い出してもらうことから始めよう（**図18.1**）．それでは，DFT値の計算式から信号解析DFが得られるプロセスを簡単に示しておく．

いま，周波数f_ℓ［Hz］，すなわち，

$$f_\ell = \ell\Delta f = \ell\times\left(\frac{f_s}{N}\right) = \ell\times\left(\frac{1}{NT}\right) \quad\cdots\cdots(1)$$

ただし，$\ell = 0, 1, 2, \ldots, (N-1)$

T［秒］はサンプリング間隔

$f_s\left(=\dfrac{1}{T}\right)$ はサンプリング周波数（単位［Hz］）

におけるDFT値$\{G_\ell\}_{\ell=0}^{\ell=N-1}$は，

$$G_\ell = \frac{1}{N}\left\{x_0 + x_1 W_N^{\ell} + \ldots + x_{N-2}W_N^{(N-2)\ell} + x_{N-1}W_N^{(N-1)\ell}\right\} \quad\cdots\cdots(2)$$

で与えられる．さらに式(2)の両辺にW_N^{ℓ}を乗じて，

$$G_\ell W_N^{\ell} = \frac{1}{N}\left\{x_0 W_N^{\ell} + x_1 W_N^{2\ell} + \ldots + x_{N-2}W_N^{(N-1)\ell} + x_{N-1}W_N^{N\ell}\right\} \quad\cdots\cdots(3)$$

と表される．ここで，

$$W_N = e^{-j\frac{2\pi}{N}} = \cos\left(\frac{2\pi}{N}\right) - j\sin\left(\frac{2\pi}{N}\right) \quad\cdots\cdots(4)$$

であることから，

$$W_N^{N} = \left(e^{-j\frac{2\pi}{N}}\right)^N = e^{-j2\pi} = 1 \quad\cdots\cdots(5)$$

という性質を適用すれば，

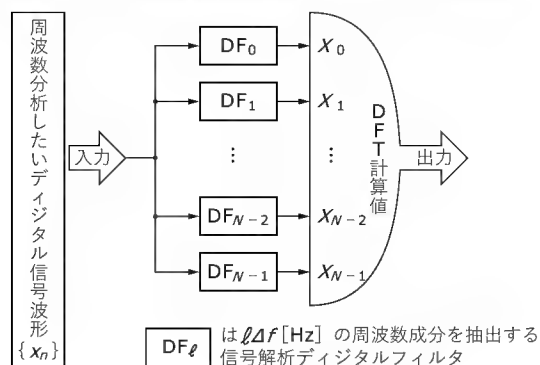
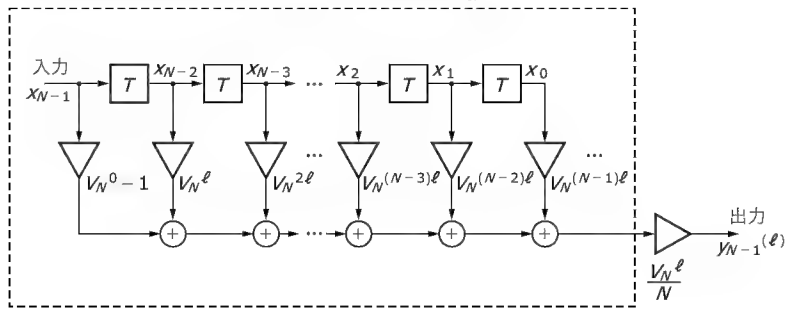
$$W_N^{(N-k)\ell} = W_N^{N\ell}\times W_N^{-k\ell} = \underbrace{\left(W_N^{N}\right)}^{\ell}\times W_N^{-k\ell} = W_N^{-k\ell} \quad\cdots\cdots(6)$$

となる関係が導かれる．

よって，式(4)，式(5)を式(3)に代入し，式(6)を適用することにより，

$$G_\ell W_N^{\ell} = \frac{1}{N}\left\{x_0 W_N^{-(N-1)\ell} + x_1 W_N^{-(N-2)\ell} + \ldots + x_{N-2}W_N^{-\ell} + x_{N-1}\right\} \quad\cdots\cdots(7)$$

となる．そこで，

〔図18.1〕DFT計算は信号解析ディジタルフィルタ群

〔図 18.2〕信号解析ディジタルフィルタの構成（$V_N = e^{j\frac{2\pi}{N}}$）

$x_n \rightarrow T \rightarrow x_{n-1}$	サンプリング間隔 T [秒] 遅延させるレジスタ
$x_n \rightarrow a \rightarrow ax_n$	乗算回路
$x_n{}^{(1)}, x_n{}^{(2)} \rightarrow + \rightarrow x_n{}^{(1)} + x_n{}^{(2)}$	2入力1出力加算回路

$$V_N = W_N{}^{-1} = e^{j\frac{2\pi}{N}} = \cos\left(\frac{2\pi}{N}\right) + j\sin\left(\frac{2\pi}{N}\right) \cdots\cdots\cdots\cdots\cdots (8)$$

と置けば，式(7)は，

$$G_\ell V_N{}^{-\ell} = \frac{1}{N}\left\{x_0 V_N{}^{(N-1)\ell} + x_1 V_N{}^{(N-2)\ell} + \ldots + x_{N-2} V_N{}^\ell + x_{N-1}\right\}$$

と表されるので，両辺に $V_N{}^\ell$ を乗ずることにより，

$$G_\ell = \frac{1}{N} V_N{}^\ell \left\{x_0 V_N{}^{(N-1)\ell} + x_1 V_N{}^{(N-2)\ell} + \ldots + x_{N-2} V_N{}^\ell + x_{N-1}\right\} \cdots\cdots\cdots\cdots (9)$$

となる関係が成立する．

以上の結果に基づき，式(9)はディジタルフィルタとして**図 18.2** のように構成される．ここで，**図 18.2** の▽印で表される係数(乗算係数，あるいはタップ係数ともいう)，すなわち，

$$\{\underbrace{V_N{}^0, V_N{}^\ell, V_N{}^{2\ell}, \ldots, V_N{}^{(N-1)\ell}}_{N個}\} \cdots\cdots\cdots\cdots (10)$$

は，最大振幅が 1 で周波数 f_ℓ [Hz]の複素正弦波，

$$v(t) = e^{j2\pi f_\ell t} \cdots\cdots\cdots\cdots (11)$$

を時間間隔 T[秒]ごとにサンプリングした値であり，k = 0，1，2，…，(N−1)に対して，

$$v(kT) = e^{j2\pi f_\ell kT} = e^{j2\pi\ell\left(\frac{1}{NT}\right)kT}$$
$$= e^{j\frac{2\pi}{N}k\ell} = \left(e^{j\frac{2\pi}{N}}\right)^{k\ell} = V_N{}^{k\ell}$$

となる．このとき，式(10)のタップ係数が直交基底ベクトルであることも覚えておいてほしい．

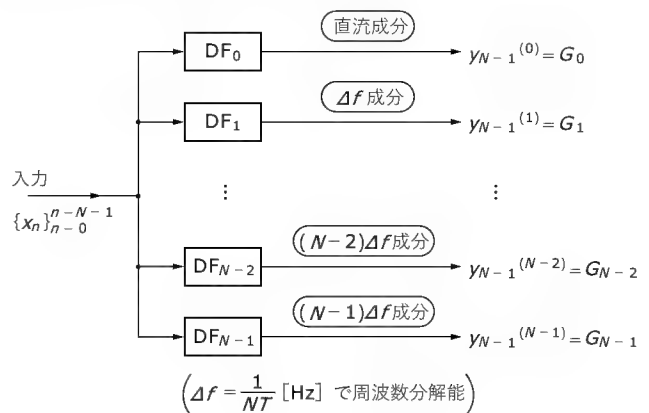
DFT のフィルタバンク構成

ところで，$t = (N-1)T$[秒]における**図 18.2** のディジタルフィルタの出力 $y_{N-1}{}^{(\ell)}$ は，

$$y_{N-1}{}^{(\ell)} = \frac{1}{N} V_N{}^\ell \left\{x_{N-1} + x_{N-2} V_N{}^\ell + \ldots + x_1 V_N{}^{(N-2)\ell} + x_0 V_N{}^{(N-1)\ell}\right\} \cdots\cdots\cdots\cdots (12)$$

であり，式(2)より，

$$y_{N-1}{}^{(\ell)} = G_\ell \cdots\cdots\cdots\cdots (13)$$

〔図 18.3〕DFT のフィルタバンク構成

となるので，DFT 値を算出できることがわかる．つまり，DFT 計算がフィルタとして実現可能であり，周波数帯域ごとのスペクトル成分を抽出する働きを有することになるのである．この周波数成分を取り出すための ℓ = 0，1，2，…，(N−1)に対する N 個の信号解析 DF は，“フィルタバンク”と称されるフィルタ群を構成する(**図 18.3**)．フィルタバンクの各出力 $\left\{y_{N-1}{}^{(\ell)}\right\}_{\ell=0}^{\ell=N-1}$ が DFT 値 $\left\{G_\ell\right\}_{\ell=0}^{\ell=N-1}$ に相当し，信号解析 DF と DFT の相互関係を知ることができよう．

次に，$t = nT$ に対して，式(12)を総和記号(Σ)を用いて表現すれば，

$$y_n{}^{(\ell)} = V_N{}^\ell \times \left[\frac{1}{N}\sum_{k=0}^{N-1} x_{n-k} V_N{}^{k\ell}\right] \cdots\cdots\cdots\cdots (14)$$

と表される“差分方程式”で記述される．式(14)の総和記号(Σ)をはずして表してみると，

$$y_n{}^{(\ell)} = V_N{}^\ell \times \left[\frac{1}{N}\left\{x_n + x_{n-1} V_N{}^\ell + x_{n-2} V_N{}^{2\ell} + \ldots + x_{n-(N-2)} V_N{}^{(N-2)\ell} + x_{n-(N-1)} V_N{}^{(N-1)\ell}\right\}\right] \cdots\cdots (15)$$

となる．続いて，両辺を z 変換して，

$$Y_\ell(z) = \sum_{n=0}^{N-1} y_n{}^{(\ell)} z^{-n}$$
$$= y_0{}^{(\ell)} + y_1{}^{(\ell)} z^{-1} + y_2{}^{(\ell)} z^{-2} + \ldots + y_{N-1}{}^{(\ell)} z^{-(N-1)}$$

$$X(z)=\sum_{k=0}^{N-1} x_k z^{-k}$$
$$=x_0+x_1z^{-1}+x_2z^{-2}+\ldots+x_{N-1}z^{-(N-1)}$$

とおけば，

$$Y_\ell(z)=V_N{}^\ell\times\left[\frac{1}{N}\left\{X(z)+z^{-1}X(z)V_N{}^\ell+z^{-2}X(z)V_N{}^{2\ell}+\ldots\right.\right.$$
$$\left.\left.+z^{-(N-2)}X(z)V_N{}^{(N-2)\ell}+z^{-(N-1)}X(z)V_N{}^{(N-1)\ell}\right\}\right] \quad \cdots\cdots(16)$$

と表される(**図 18.4**)．なお，式(3)の差分方程式をもとに，z 変換して伝達関数を得る計算プロセスは，誌面の都合で割愛する．詳細については，拙著『やり直しのための工業数学』(CQ出版社)の第19章「z 変換」に詳述してあるので，参考としてお薦めしたい．

次に，式(16)より，信号解析 DF の伝達関数 $H_\ell(z)$ は，

$$H_\ell(z)=\frac{Y_\ell(z)}{X(z)}\left(=\frac{出力信号の z 変換}{入力信号の z 変換}\right)$$

$$=V_N{}^\ell\times\left[\frac{1}{N}\left\{1+z^{-1}V_N{}^\ell+z^{-2}V_N{}^{2\ell}\ldots+z^{-(N-1)}V_N{}^{(N-1)\ell}\right\}\right] \quad \cdots(17)$$

と書ける．ここで，$V_N{}^\ell$ は入力信号のタイミングの違いと出力を整理するための処理であり，位相進みに相当する〔ポリフェーズ処理とよばれる．ポリ(poly)は「複数」，フェーズ(phase)は「位相」の意味〕．

さて，式(17)の進み位相分 $V_N{}^\ell$ を除いた伝達関数 $\tilde{H}_\ell(z)$ を，

$$\tilde{H}_\ell(z)=\frac{1}{N}\left\{1+z^{-1}V_N{}^\ell+z^{-2}V_N{}^{2\ell}\ldots+z^{-(N-1)}V_N{}^{(N-1)\ell}\right\} \quad \cdots(18)$$

と定義するとき，

$$H_\ell(z)=V_N{}^\ell\tilde{H}_\ell(z) \quad \cdots\cdots(19)$$

と表される(**図 18.5**)．なお，伝達関数 $\tilde{H}_\ell(z)$ の添字 ℓ は式(1)の DFT 計算で分析できる周波数 f_ℓ [Hz]である．

以上のように，ディジタル信号 $\{x_n\}_{n=k-N+1}^{n=k}$ に対して，$\Delta f=\dfrac{1}{NT}$ [Hz]ごとの離散周波数でのスペクトル値が，**図 18.3** のフィルタバンク構成における N 個の信号解析 DF の出力値として得られることが理解できる．

例題1

$N=4$ サンプルに対する DFT のフィルタバンク構成のブロック図を示せ．あわせて，各信号解析 DF の差分方程式，伝達関数も求めよ．

解答1

式(8)より，

$$V_4=e^{j\frac{2\pi}{4}}=\cos\left(\frac{2\pi}{4}\right)+j\sin\left(\frac{2\pi}{4}\right)=j$$

$$\begin{cases} V_4{}^0=V_4{}^4=V_4{}^8=\ldots=1 \\ V_4{}^1=V_4{}^5=V_4{}^9=\ldots=j \\ V_4{}^2=V_4{}^6=V_4{}^{10}=\ldots=-1 \\ V_4{}^3=V_4{}^7=V_4{}^{11}=\ldots=-j \end{cases} \quad \cdots\cdots(20)$$

となる関係が成り立つことに注意して，式(15)に基づき，信号解析 DF の差分方程式が次のように得られる．

$$y_n{}^{(0)}=\frac{1}{4}\left\{x_n+x_{n-1}+x_{n-2}+x_{n-3}\right\} \quad \cdots\cdots(21)$$

$$y_n{}^{(1)}=V_4{}^1\left[\frac{1}{4}\left\{x_n+x_{n-1}V_4{}^1+x_{n-2}V_4{}^2+x_{n-3}V_4{}^3\right\}\right]$$
$$=j\times\left[\frac{1}{4}\left\{x_n+jx_{n-1}-x_{n-2}-jx_{n-3}\right\}\right] \quad \cdots\cdots(22)$$

$$y_n{}^{(2)}=V_4{}^2\left[\frac{1}{4}\left\{x_n+x_{n-1}V_4{}^2+x_{n-2}V_4{}^4+x_{n-3}V_4{}^6\right\}\right]$$
$$=(-1)\times\left[\frac{1}{4}\left\{x_n-x_{n-1}+x_{n-2}-x_{n-3}\right\}\right] \quad \cdots\cdots(23)$$

$$y_n{}^{(3)}=V_N{}^3\left[\frac{1}{4}\left\{x_n+x_{n-1}V_4{}^3+x_{n-2}V_4{}^6+x_{n-3}V_4{}^9\right\}\right]$$
$$=(-j)\times\left[\frac{1}{4}\left\{x_n-jx_{n-1}-x_{n-2}+jx_{n-3}\right\}\right] \quad \cdots\cdots(24)$$

また，式(18)，式(19)より伝達関数は次のように表される．

〔図 18.4〕信号解析ディジタルフィルタの伝達関数計算の流れ

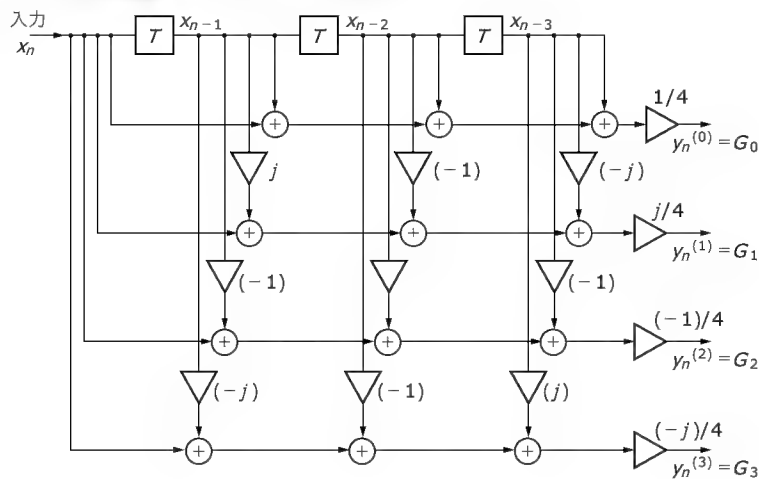
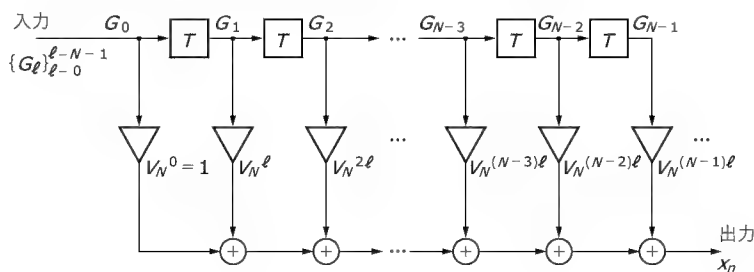
(**a**) 差分方程式の z 変換

$$H_\ell(z)=\frac{Y_\ell(z)}{X(z)}=V_N{}^\ell\times\left[\frac{1}{N}\{1+z^{-1}V_N{}^\ell+z^{-2}V_N{}^{2\ell}+\cdots+z^{-(N-1)}V_N{}^{(N-1)\ell}\}\right]$$

(**b**) 伝達関数

〔図 18.5〕DFT のフィルタバンク構成におけるポリフェーズ処理

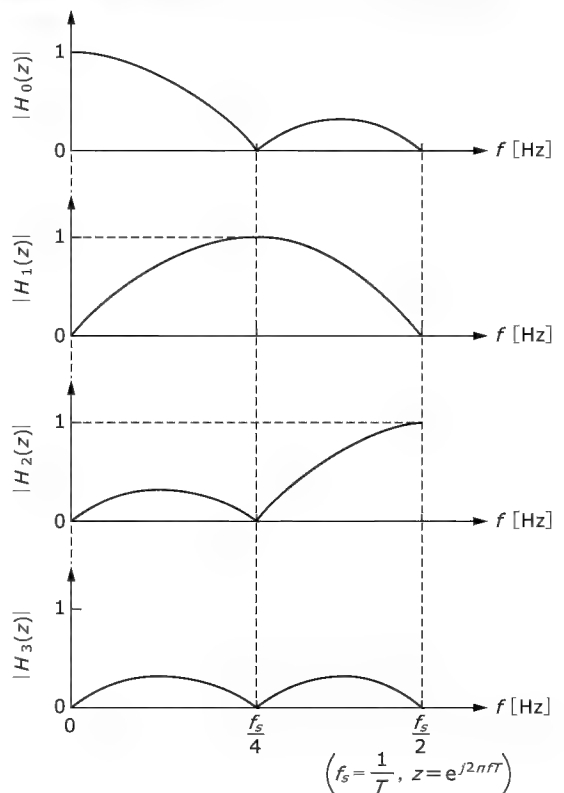
〔図18.6〕例題1のフィルタバンク構成

〔図18.7〕例題1の各信号解析ディジタルフィルタの振幅特性

〔図18.8〕合成ディジタルフィルタの構成

$$\begin{cases}\tilde{H}_0(z)=\dfrac{1}{4}\left(1+z^{-1}+z^{-2}+z^{-3}\right)\\ H_0(z)=\tilde{H}_0(z)\end{cases}\quad\cdots\cdots(25)$$

$$\begin{cases}\tilde{H}_1(z)=\dfrac{1}{4}\left(1+z^{-1}V_4^{\,1}+z^{-2}V_4^{\,2}+z^{-3}V_4^{\,3}\right)\\ \qquad\quad=\dfrac{1}{4}\left(1+jz^{-1}-z^{-2}-jz^{-3}\right)\\ H_1(z)=V_4^{\,1}\tilde{H}_1(z)=j\tilde{H}_1(z)\end{cases}\quad\cdots\cdots(26)$$

$$\begin{cases}\tilde{H}_2(z)=\dfrac{1}{4}\left(1+z^{-1}V_4^{\,2}+z^{-2}V_4^{\,4}+z^{-3}V_4^{\,6}\right)\\ \qquad\quad=\dfrac{1}{4}\left(1-z^{-1}+z^{-2}-z^{-3}\right)\\ H_2(z)=V_4^{\,2}\tilde{H}_2(z)=-\tilde{H}_2(z)\end{cases}\quad\cdots\cdots(27)$$

$$\begin{cases}\tilde{H}_3(z)=\dfrac{1}{4}\left(1+z^{-1}V_4^{\,3}+z^{-2}V_4^{\,6}+z^{-3}V_4^{\,9}\right)\\ \qquad\quad=\dfrac{1}{4}\left(1-jz^{-1}-z^{-2}+jz^{-3}\right)\\ H_3(z)=V_4^{\,3}\tilde{H}_3(z)=-j\tilde{H}_3(z)\end{cases}\quad\cdots\cdots(28)$$

以上の結果より導き出されるフィルタバンク構成を**図18.6**に示す．また，式(25)～式(28)の各信号解析DFの振幅特性は，$z=e^{j2\pi fT}$を代入して絶対値を計算することにより，

$$\begin{aligned}\left|H_\ell\left(e^{j2\pi fT}\right)\right|&=\left|V_4^{\,\ell}\tilde{H}_\ell\left(e^{j2\pi fT}\right)\right|\\&=\underbrace{\left|V_4^{\,\ell}\right|}_{1}\left|\tilde{H}_\ell\left(e^{j2\pi fT}\right)\right|=\left|\tilde{H}_\ell\left(e^{j2\pi fT}\right)\right|\end{aligned}\quad\cdots\cdots(29)$$

で求められる(**図18.7**)．

図18.7より，各信号解析DFの振幅特性の特徴は，

$\tilde{H}_0(z)$ ：ローパス(低域通過)フィルタ

$\tilde{H}_1(z)$ ：バンドパス(帯域通過)フィルタ

$\tilde{H}_2(z)$ ：ハイパス(高域通過)フィルタ

$\tilde{H}_3(z)$ ：バンドエリミネーション(帯域阻止)フィルタ

であり，DFT値の計算が周波数成分ごとに信号分解するためのフィルタバンクと等価なシステムであることもわかる．

IDFTのフィルタバンク構成

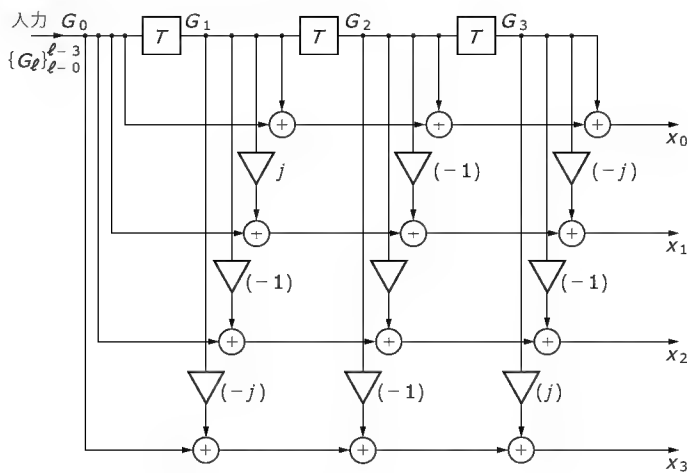
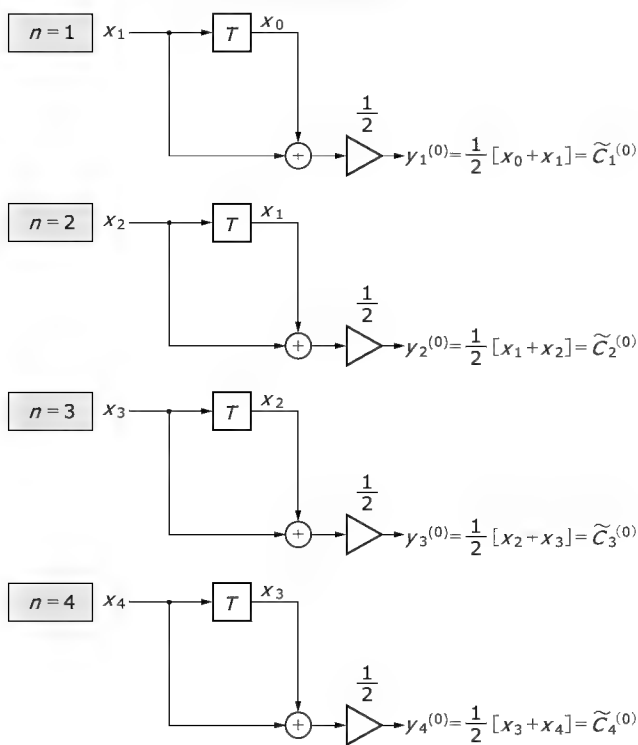
ここでは，DFTの逆変換であるIDFTも，DFTと同様にフィルタバンク構成できることを示す．

いま，N個の周波数成分であるDFT値を$\left\{G_\ell\right\}_{\ell=0}^{\ell=N-1}$とすれば，IDFTは$n=0,\ 1,\ 2,\ \ldots,\ (N-1)$に対して，

$$x_n=G_0+G_1W_N^{-n}+G_2W_N^{-2n}+\ldots+G_{N-1}W_N^{-(N-1)n}\quad\cdots\cdots(30)$$

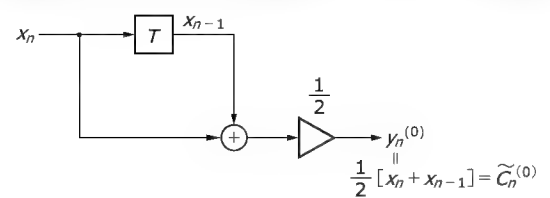
と与えられる．ここで，式(8)の関係から，式(30)は，

$$x_n=G_0+G_1V_N^{\,n}+G_2V_N^{\,2n}+\ldots+G_{N-1}V_N^{\,(N-1)n}\quad\cdots\cdots(31)$$

〔**図 18.9**〕**IDFT のフィルタバンク構成**

〔**図 18.11**〕**DCT 値 $C_0^{(2)}$ の計算処理の流れ**

と別表現され，ディジタルシステムとして実現できる(**図 18.8**，前頁)．**図 18.8** のディジタルシステムは元のディジタル信号を再合成する働きをすることから，ここでは合成ディジタルフィルタ(以後，合成 DF と記す)と呼ぶことにする．

また，**図 18.8** の合成 DF の構成は**図 18.2** の信号解析 DF の構成に示す点線内のブロックと同じものになっていることもわかる．

一例として，$N = 4$ サンプルに対する IDFT のフィルタバンク構成を示す(**図 18.9**)．なお，**図 18.9** の各合成 DF の出力は，式(31)に基づき，式(20)を適用して，

〔**図 18.10**〕**DCT 値 $C_0^{(2)}$ の算出ディジタルフィルタの構成**

$$\begin{cases} x_0 = G_0 + G_1 + G_2 + G_3 \\ x_1 = G_0 + G_1V_4^1 + G_2V_4^2 + G_3V_4^3 \\ \quad = G_0 + jG_1 - G_2 - jG_3 \\ x_2 = G_0 + G_1V_4^2 + G_2V_4^4 + G_3V_4^6 \\ \quad = G_0 - G_1 + G_2 - G_3 \\ x_3 = G_0 + G_1V_4^3 + G_2V_4^6 + G_3V_4^9 \\ \quad = G_0 - jG_1 - G_2 + jG_3 \end{cases} \quad \cdots\cdots (32)$$

と表される．

DCT の分析フィルタバンク構成

手始めに DCT とディジタルフィルタとの相互関係をイメージしてもらうことを目的として，$N = 2$ サンプルのディジタル信号 $\left\{x_n\right\}_{n=0}^{n=1}$ に対する DCT を例に説明する．

まず，DCT は次式で与えられる．

$$C_0^{(2)} = \frac{1}{2}\left[x_0 + x_1\right] \quad \cdots\cdots (33)$$

$$C_1^{(2)} = \frac{1}{2}\left[x_0\left\{\sqrt{2}\cos\left(\frac{\pi}{4}\right)\right\} + x_1\left\{\sqrt{2}\cos\left(\frac{3\pi}{4}\right)\right\}\right]$$
$$= \frac{1}{2}\left[x_0 - x_1\right] \quad \cdots\cdots (34)$$

ところで，式(33)はディジタルフィルタとして**図 18.10** のように構成され，出力 $y_n^{(0)}$ は，

$$y_n^{(0)} = \frac{1}{2}\left\{x_n + x_{n-1}\right\} \quad \cdots\cdots (35)$$

であり，

$$y_n^{(0)} = \tilde{C}_n^{(0)} \quad \cdots\cdots (36)$$

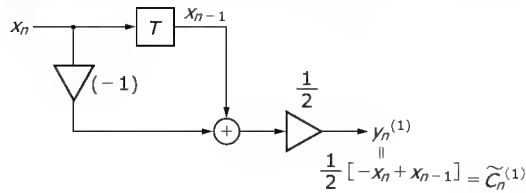
となるので $\ell = 0$ に対する DCT 値を算出できることがわかる(**図 18.11**)．ただし，$\tilde{C}_n^{(0)}$ はディジタル信号 $\left\{x_n,\ x_{n-1}\right\}$ の $\ell = 0$ の DCT 値を意味しており，$n = 1$ に対する出力値は，

$$y_1^{(0)} = \frac{1}{2}\left\{x_1 + x_0\right\} = \tilde{C}_1^{(0)} = C_0^{(2)} \quad \cdots\cdots (37)$$

であるので，式(33)の DCT 値が得られることになる．

さらに続けて，$n = 2,\ 3,\ 4,\ \ldots$ と計算を進めていくと，

$$\begin{cases} y_2^{(0)} = \frac{1}{2}\left\{x_2 + x_1\right\} = \tilde{C}_2^{(0)} \\ y_3^{(0)} = \frac{1}{2}\left\{x_3 + x_2\right\} = \tilde{C}_3^{(0)} \\ y_4^{(0)} = \frac{1}{2}\left\{x_4 + x_3\right\} = \tilde{C}_4^{(0)} \\ \quad \vdots \end{cases} \quad \cdots\cdots (38)$$

〔図 18.12〕DCT 値 $C_1^{(2)}$ の算出ディジタルフィルタの構成

となり，DCT 値が次々にフィルタ出力されることがわかる．

同様に，式(34)はディジタルフィルタとして**図 18.12** のように構成され，出力 $y_n^{(1)}$ は，

$$y_n^{(1)} = \frac{1}{2}\left\{-x_n + x_{n-1}\right\} = (-1)\times\frac{1}{2}\left\{x_n - x_{n-1}\right\} \quad \cdots\cdots\cdots\cdots(39)$$

であり，

$$y_n^{(1)} = \tilde{C}_n^{(1)} \quad \cdots\cdots\cdots\cdots(40)$$

となるので $\ell = 1$ に対する DCT 値を算出できることがわかる（**図 18.13**）．ただし，$\tilde{C}_n^{(1)}$ はディジタル信号 $\left\{x_n, x_{n-1}\right\}$ の $\ell = 1$ の DCT 値を意味しており，$n = 1$ に対する出力値は，

$$y_1^{(1)} = (-1)\times\frac{1}{2}\left\{x_1 - x_0\right\} = \tilde{C}_1^{(1)} = C_0^{(2)} \quad \cdots\cdots\cdots\cdots(41)$$

であるので，式(34)の DCT 値が得られることになる．

さらに続けて，$n = 2, 3, 4, \ldots$ と計算を進めていくと，

$$\begin{cases} y_2^{(1)} = (-1)\times\frac{1}{2}\left\{x_2 - x_1\right\} = \tilde{C}_2^{(1)} \\ y_3^{(1)} = (-1)\times\frac{1}{2}\left\{x_3 - x_2\right\} = \tilde{C}_3^{(1)} \\ y_4^{(1)} = (-1)\times\frac{1}{2}\left\{x_4 - x_3\right\} = \tilde{C}_4^{(1)} \\ \quad\vdots \end{cases} \quad \cdots\cdots\cdots\cdots(42)$$

となり，DCT 値が次々にフィルタ出力されることがわかる．

ここで，式(35)，式(39)の差分方程式の両辺を z 変換して DCT 計算のディジタルフィルタの伝達関数はそれぞれ，

$$H_0(z) = \frac{1}{2}\left(1 + z^{-1}\right) \quad \cdots\cdots\cdots\cdots(43)$$

$$H_1(z) = (-1)\times\left[\frac{1}{2}\left(1 - z^{-1}\right)\right] \quad \cdots\cdots\cdots\cdots(44)$$

と表される．

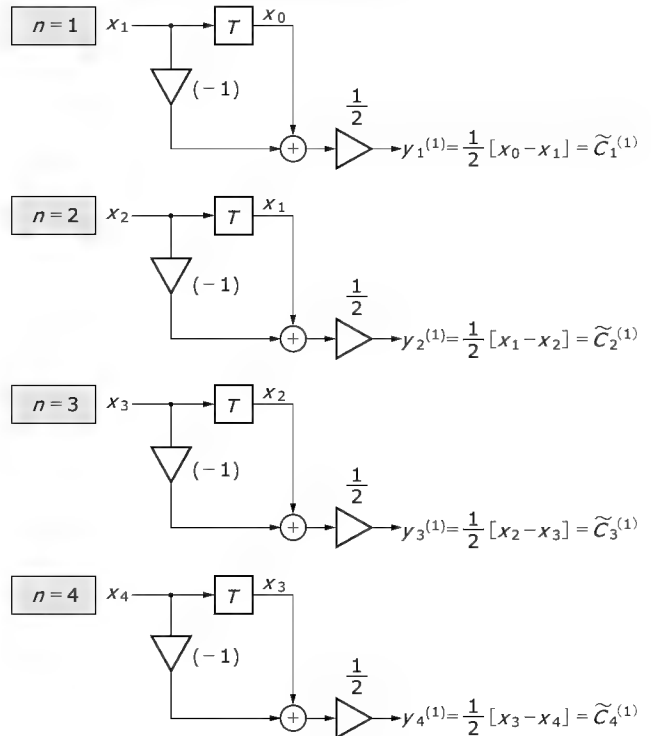
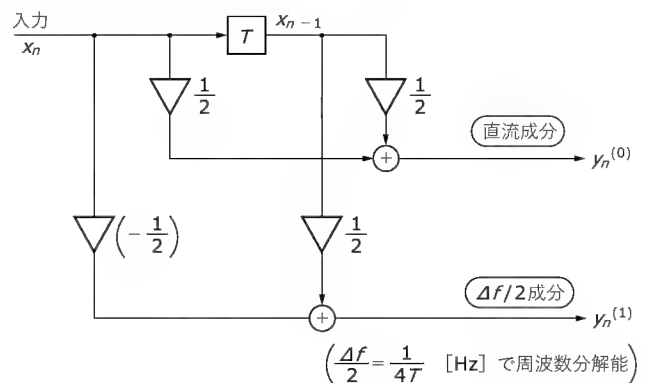
以上より，式(43)，式(44)の伝達関数の(　)の中の z に関する多項式の係数値が，それぞれ直交基底ベクトル，すなわち，

$$\boldsymbol{\phi}^{(0)} = \left\{1, 1\right\}$$

$$\boldsymbol{\phi}^{(1)} = \left\{\sqrt{2}\cos\left(\frac{\pi}{4}\right), \sqrt{2}\cos\left(\frac{3\pi}{4}\right)\right\} = \left\{1, -1\right\}$$

の各要素値に一致し，しかも**図 18.10** と**図 18.12** におけるディジタルフィルタのタップ係数になるのである．もちろん，式(43)，式(44)を整理して，

$$H_0(z) = \frac{1}{2} + \frac{1}{2}z^{-1} \quad \cdots\cdots\cdots\cdots(45)$$

〔図 18.13〕DCT 値 $C_1^{(2)}$ の計算処理の流れ

〔図 18.14〕DCT のフィルタバンク構成（$N = 2$）

$$H_1(z) = -\frac{1}{2} + \frac{1}{2}z^{-1} \quad \cdots\cdots\cdots\cdots(46)$$

と表すことにより，**図 18.14** のようにフィルタバンクを構成することができる．

次に，**図 18.14** の二つのディジタルフィルタの振幅特性を計算しておこう．振幅特性は，伝達関数において $z = e^{j2\pi fT}$ を代入して絶対値を計算すればよく，

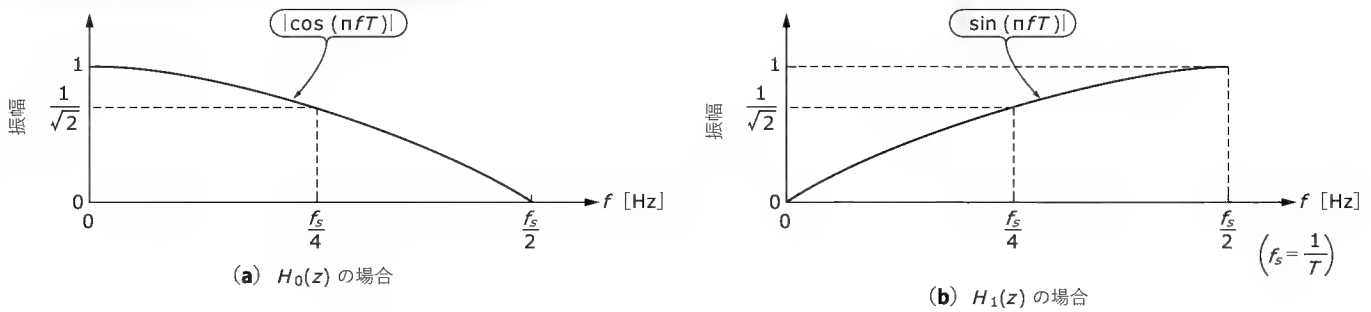
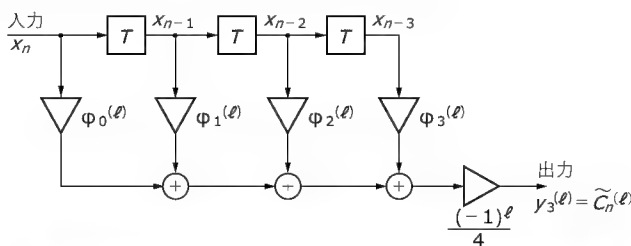
$$\left|H_0\left(e^{j2\pi fT}\right)\right| = \left|\cos(\pi fT)\right| \quad \cdots\cdots\cdots\cdots(47)$$

$$\left|H_1\left(e^{j2\pi fT}\right)\right| = \left|\sin(\pi fT)\right| \quad \cdots\cdots\cdots\cdots(48)$$

と求められる（**図 18.15**）．**図 18.15**(**a**)より，$\ell = 0$ に対する

〔図 18.15〕各分析ディジタルフィルタの振幅特性（$N = 2$）

(a) $H_0(z)$ の場合

(b) $H_1(z)$ の場合

〔図 18.16〕分析ディジタルフィルタの構成（$N = 4$）

$\ell = 0$ のとき，$\varphi_0^{(0)} = \varphi_1^{(0)} = \varphi_2^{(0)} = \varphi_3^{(0)} = 1$

$\ell \neq 0$ のとき，$\varphi_0^{(\ell)} = \sqrt{2}\cos\left(\frac{\ell\pi}{8}\right)$，$\varphi_1^{(\ell)} = \sqrt{2}\cos\left(\frac{3\ell\pi}{8}\right)$

$\varphi_2^{(\ell)} = \sqrt{2}\cos\left(\frac{5\ell\pi}{8}\right)$，$\varphi_3^{(\ell)} = \sqrt{2}\cos\left(\frac{7\ell\pi}{8}\right)$

DCT 値を出力する振幅特性 $\left|H_0\left(e^{j2\pi fT}\right)\right|$ は低い周波数成分が通りやすい“ローパスフィルタ”，他方 $\ell = 1$ に対する DCT 値を出力する振幅特性 $\left|H_1\left(e^{j2\pi fT}\right)\right|$ は**図 18.15(b)** より，高い周波数成分が通りやすい“ハイパスフィルタ”であることがわかる．

したがって，DCT 計算を実現する**図 18.14** のフィルタバンクは周波数帯域ごとのスペクトル成分を抽出する働きを有しており，ここでは分析ディジタルフィルタ（以後，分析 DF と記す）と呼ぶことにする．なお，分析フィルタバンクの出力（DCT 値）はサブバンド信号とよばれ，応用によってさまざまな処理が施される．また，同じサンプル数に対して，DCT の周波数分解能が DFT の 2 倍に細かくなること〔式(1)中の Δf の半分，すなわち $\Delta f/2$〕も記憶に留めておいてもらいたい．

例題2

式(47)，(48)の振幅特性を，式(45)，式(46)より計算して導き出せ．

解答2

$$H_0\left(e^{j2\pi fT}\right) = \frac{1 + e^{-j2\pi fT}}{2} = e^{-j\pi fT} \times \frac{e^{\pi fT} + e^{-j\pi fT}}{2}$$

$$= e^{-j\pi fT} \times \frac{\{\cos(\pi fT) + j\sin(\pi fT)\} + \{\cos(\pi fT) - j\sin(\pi fT)\}}{2}$$

$$= e^{-j\pi fT} \times \cos(\pi fT)$$

よって，上式の絶対値を採って，$\left|e^{-j\pi fT}\right| = 1$ であることを考慮すれば，

$$\left|H_0\left(e^{j2\pi fT}\right)\right| = \underbrace{\left|e^{-j\pi fT}\right|}_{1} \times \left|\cos(\pi fT)\right| = \left|\cos(\pi fT)\right|$$

となる．同様に，

$$H_1\left(e^{j2\pi fT}\right) = \frac{1 - e^{-j2\pi fT}}{2} = e^{-j\pi fT} \times \frac{e^{\pi fT} - e^{-j\pi fT}}{2}$$

$$= e^{-j\pi fT} \times \frac{\{\cos(\pi fT) + j\sin(\pi fT)\} - \{\cos(\pi fT) - j\sin(\pi fT)\}}{2}$$

$$= je^{-j\pi fT} \times \sin(\pi fT)$$

となり，上式の絶対値を採って，$|j| = 1$，$\left|e^{-j\pi fT}\right| = 1$ であることを考慮すれば，

$$\left|H_1\left(e^{j2\pi fT}\right)\right| = \underbrace{|j|}_{1} \times \underbrace{\left|e^{-j\pi fT}\right|}_{1} \times \left|\sin(\pi fT)\right| = \left|\sin(\pi fT)\right|$$

という振幅特性が導かれる．

次は，少々本格的に $N = 4$ サンプルのディジタル信号 $\{x_n\}_{n=0}^{n=3}$ に対する DCT 計算を考えてみたい．DCT 値は，次式で与えられる．

$$C_0^{(4)} = \frac{1}{4}\left[x_0 + x_1 + x_2 + x_3\right] \quad \cdots\cdots (49)$$

$$C_1^{(4)} = \frac{1}{4}\left[x_0\left\{\sqrt{2}\cos\left(\frac{\pi}{8}\right)\right\} + x_1\left\{\sqrt{2}\cos\left(\frac{3\pi}{8}\right)\right\} + x_2\left\{\sqrt{2}\cos\left(\frac{5\pi}{8}\right)\right\} + x_3\left\{\sqrt{2}\cos\left(\frac{7\pi}{8}\right)\right\}\right] \quad \cdots\cdots (50)$$

$$C_2^{(4)} = \frac{1}{4}\left[x_0\left\{\sqrt{2}\cos\left(\frac{2\pi}{8}\right)\right\} + x_1\left\{\sqrt{2}\cos\left(\frac{6\pi}{8}\right)\right\} + x_2\left\{\sqrt{2}\cos\left(\frac{10\pi}{8}\right)\right\} + x_3\left\{\sqrt{2}\cos\left(\frac{14\pi}{8}\right)\right\}\right]$$

$$= \frac{1}{4}\left[x_0 - x_1 - x_2 + x_3\right] \quad \cdots\cdots (51)$$

$$C_3^{(4)} = \frac{1}{4}\left[x_0\left\{\sqrt{2}\cos\left(\frac{3\pi}{8}\right)\right\} + x_1\left\{\sqrt{2}\cos\left(\frac{9\pi}{8}\right)\right\} + x_2\left\{\sqrt{2}\cos\left(\frac{15\pi}{8}\right)\right\} + x_3\left\{\sqrt{2}\cos\left(\frac{21\pi}{8}\right)\right\}\right] \quad \cdots\cdots (52)$$

このとき，式(49)～式(52)の{ }の中は直交基底ベクトルの各要素値，すなわち，

$$
\begin{cases}
\boldsymbol{\phi}^{(0)} = \left\{1,\ 1,\ 1,\ 1\right\} \\
\boldsymbol{\phi}^{(1)} = \left\{\sqrt{2}\cos\left(\dfrac{\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{3\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{5\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{7\pi}{8}\right)\right\} \\
\boldsymbol{\phi}^{(2)} = \left\{\sqrt{2}\cos\left(\dfrac{2\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{6\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{9\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{14\pi}{8}\right)\right\} \\
\boldsymbol{\phi}^{(3)} = \left\{\sqrt{2}\cos\left(\dfrac{3\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{9\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{15\pi}{8}\right),\ \sqrt{2}\cos\left(\dfrac{21\pi}{8}\right)\right\}
\end{cases}
\quad\cdots\cdots\cdots\cdots(53)
$$

に一致していることに気づかれた人もおられると思う．よって，直交基底ベクトル $\varphi^{(\ell)}$ について，各要素値を，

$$\boldsymbol{\phi}^{(\ell)} = \left\{\phi_0^{(\ell)}, \phi_1^{(\ell)}, \phi_2^{(\ell)}, \phi_3^{(\ell)}\right\} \quad\cdots\cdots\cdots\cdots(54)$$

と表せば，DCT値を出力するディジタルフィルタの構成は**図18.16**のようになり，出力 $y_n^{(\ell)}$ は，

$$y_n^{(\ell)} = (-1)^\ell \times \left[\frac{1}{4}\left\{\phi_0^{(\ell)} x_n + \phi_1^{(\ell)} x_{n-1} + \phi_2^{(\ell)} x_{n-2} + \phi_3^{(\ell)} x_{n-3}\right\}\right] \quad\cdots\cdots\cdots\cdots(55)$$

であり，

$$y_n^{(\ell)} = \tilde{C}_n^{(\ell)} \quad\cdots\cdots\cdots\cdots(56)$$

となるので，ℓ に対するDCT値を算出できることがわかる(**図18.17**)．ただし，$\tilde{C}_n^{(\ell)}$ は $t = nT$[秒]における4サンプルのディジタル信号 $\left\{x_n, x_{n-1}, x_{n-2}, x_{n-3}\right\}$ のDCT値を表しており，$n = 3$ に対する出力値は，

$$
\begin{aligned}
y_3^{(\ell)} &= (-1)^\ell \times \left[\frac{1}{4}\left\{\phi_0^{(\ell)} x_3 + \phi_1^{(\ell)} x_2 + \phi_2^{(\ell)} x_1 + \phi_3^{(\ell)} x_0\right\}\right] \\
&= \tilde{C}_3^{(\ell)} = C_\ell^{(4)} \quad\cdots\cdots\cdots\cdots(57)
\end{aligned}
$$

であるので，式(49)～式(52)のDCT値が得られることになる．

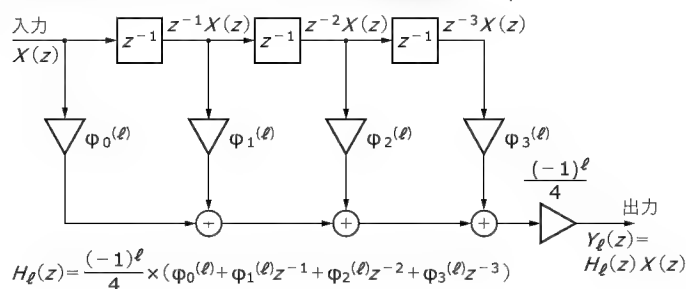
さらに続けて，$n = 4,\ 5,\ 6,\ \ldots$と計算を進めていくと，

$$
\begin{cases}
y_4^{(\ell)} = (-1)^\ell \times \left[\dfrac{1}{4}\left\{\phi_0^{(\ell)} x_4 + \phi_1^{(\ell)} x_3 + \phi_2^{(\ell)} x_2 + \phi_3^{(\ell)} x_1\right\}\right] = \tilde{C}_4^{(\ell)} \\
y_5^{(\ell)} = (-1)^\ell \times \left[\dfrac{1}{4}\left\{\phi_0^{(\ell)} x_5 + \phi_1^{(\ell)} x_4 + \phi_2^{(\ell)} x_3 + \phi_3^{(\ell)} x_2\right\}\right] = \tilde{C}_5^{(\ell)} \\
y_6^{(\ell)} = (-1)^\ell \times \left[\dfrac{1}{4}\left\{\phi_0^{(\ell)} x_6 + \phi_1^{(\ell)} x_5 + \phi_2^{(\ell)} x_4 + \phi_3^{(\ell)} x_3\right\}\right] = \tilde{C}_6^{(\ell)} \\
\quad\vdots
\end{cases}
\quad\cdots\cdots\cdots\cdots(58)
$$

となり，DCT値が次々にフィルタ出力されることがわかる．

次に，式(55)の差分方程式の両辺を z 変換してDCT計算のディジタルフィルタの伝達関数は，

$$H_\ell(z) = (-1)^\ell \times \left[\frac{1}{4}\left(\phi_0^{(\ell)} + \phi_1^{(\ell)} z^{-1} + \phi_2^{(\ell)} z^{-2} + \phi_3^{(\ell)} z^{-3}\right)\right] \quad\cdots\cdots\cdots\cdots(59)$$

〔図18.17〕分析ディジタルフィルタの伝達関数($N = 4$)

と表される．

以上より，例題3の各分析DFの伝達関数の{ }の中の z に関する多項式の係数値が，式(53)の直交基底ベクトル $\left\{\boldsymbol{\phi}^{(\ell)}\right\}_{\ell=0}^{\ell=3}$ の各要素値から算出された値に一致し，しかも**図18.17**におけるディジタルフィルタのタップ係数になる．

例題3

$N = 4$ サンプルのディジタル信号のDCT値を出力する分析フィルタバンクの全体構成，および各分析DFの振幅特性を示せ．

解答3

式(53)，式(59)に基づき，分析DFの伝達関数 $\left\{H_\ell(z)\right\}_{\ell=0}^{\ell=3}$ を計算する．

$$H_0(z) = \frac{1}{4}\left(1 + z^{-1} + z^{-2} + z^{-3}\right)$$

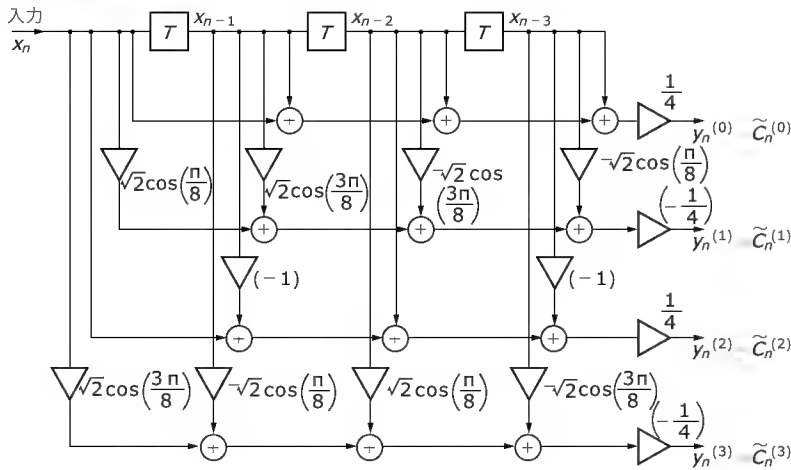
$$
\begin{aligned}
H_1(z) &= (-1) \times \left[\frac{\sqrt{2}}{4}\left\{\cos\left(\frac{\pi}{8}\right) + \cos\left(\frac{3\pi}{8}\right)z^{-1} + \cos\left(\frac{5\pi}{8}\right)z^{-2} + \cos\left(\frac{7\pi}{8}\right)z^{-3}\right\}\right] \\
&= \frac{\sqrt{2}}{4}\left\{-\cos\left(\frac{\pi}{8}\right) - \cos\left(\frac{3\pi}{8}\right)z^{-1} + \cos\left(\frac{3\pi}{8}\right)z^{-2} + \cos\left(\frac{\pi}{8}\right)z^{-3}\right\}
\end{aligned}
$$

$$
\begin{aligned}
H_2(z) &= \frac{\sqrt{2}}{4}\left\{\cos\left(\frac{2\pi}{8}\right) + \cos\left(\frac{6\pi}{8}\right)z^{-1} + \cos\left(\frac{10\pi}{8}\right)z^{-2} + \cos\left(\frac{14\pi}{8}\right)z^{-3}\right\} \\
&= \frac{1}{4}\left(1 - z^{-1} - z^{-2} + z^{-3}\right)
\end{aligned}
$$

$$
\begin{aligned}
H_3(z) &= (-1) \times \left[\frac{\sqrt{2}}{4}\left\{\cos\left(\frac{3\pi}{8}\right) + \cos\left(\frac{9\pi}{8}\right)z^{-1} + \cos\left(\frac{15\pi}{8}\right)z^{-2} + \cos\left(\frac{21\pi}{8}\right)z^{-3}\right\}\right] \\
&= \frac{\sqrt{2}}{4}\left\{-\cos\left(\frac{3\pi}{8}\right) + \cos\left(\frac{\pi}{8}\right)z^{-1} - \cos\left(\frac{\pi}{8}\right)z^{-2} + \cos\left(\frac{3\pi}{8}\right)z^{-3}\right\}
\end{aligned}
$$

〔図 18.18〕例題3

(a) 分析フィルタバンク

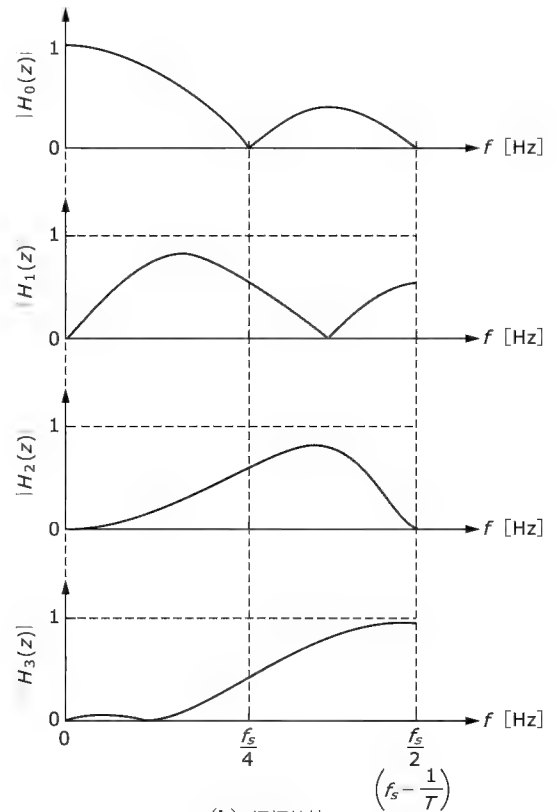
(b) 振幅特性

以上より，フィルタバンク構成を**図 18.18**(**a**)に，各分析 DF の振幅特性を**図 18.18**(**b**)に示す.

IDCT の合成フィルタバンク構成

ところで，周波数帯域ごとに分解されたスペクトル値から元のディジタル信号を再合成するには，DCT の逆変換としての IDCT を実行すればよいことは明白であろう．DCT の例示として用いた $N = 4$ サンプルの場合について，IDCT のフィルタバンク構成を説明する.

いま，DCT 値を $\left\{C_\ell^{(4)}\right\}_{\ell=0}^{\ell=3}$ とすれば，IDCT は，

$$\begin{cases} x_0 = C_0^{(4)} + \sqrt{2}\cos\left(\frac{\pi}{8}\right)C_1^{(4)} + \sqrt{2}\cos\left(\frac{2\pi}{8}\right)C_2^{(4)} \\ \qquad + \sqrt{2}\cos\left(\frac{3\pi}{8}\right)C_3^{(4)} \\ x_1 = C_0^{(4)} + \sqrt{2}\cos\left(\frac{3\pi}{8}\right)C_1^{(4)} + \sqrt{2}\cos\left(\frac{6\pi}{8}\right)C_2^{(4)} \\ \qquad + \sqrt{2}\cos\left(\frac{9\pi}{8}\right)C_3^{(4)} \\ x_2 = C_0^{(4)} + \sqrt{2}\cos\left(\frac{5\pi}{8}\right)C_1^{(4)} + \sqrt{2}\cos\left(\frac{10\pi}{8}\right)C_2^{(4)} \\ \qquad + \sqrt{2}\cos\left(\frac{15\pi}{8}\right)C_3^{(4)} \\ x_3 = C_0^{(4)} + \sqrt{2}\cos\left(\frac{7\pi}{8}\right)C_1^{(4)} + \sqrt{2}\cos\left(\frac{14\pi}{8}\right)C_2^{(4)} \\ \qquad + \sqrt{2}\cos\left(\frac{21\pi}{8}\right)C_3^{(4)} \end{cases} \quad \cdots(60)$$

と表される．また，式(53)と式(54)の直交基底ベクトルの要素値を用いると，式(60)の IDCT は，

$$\begin{cases} x_0 = \phi_0^{(0)}C_0^{(4)} + \phi_0^{(1)}C_1^{(4)} + \phi_0^{(2)}C_2^{(4)} + \phi_0^{(3)}C_3^{(4)} \\ x_1 = \phi_1^{(0)}C_0^{(4)} + \phi_1^{(1)}C_1^{(4)} + \phi_1^{(2)}C_2^{(4)} + \phi_1^{(3)}C_3^{(4)} \\ x_2 = \phi_2^{(0)}C_0^{(4)} + \phi_2^{(1)}C_1^{(4)} + \phi_2^{(2)}C_2^{(4)} + \phi_2^{(3)}C_3^{(4)} \\ x_3 = \phi_3^{(0)}C_0^{(4)} + \phi_3^{(1)}C_1^{(4)} + \phi_3^{(2)}C_2^{(4)} + \phi_3^{(3)}C_3^{(4)} \end{cases} \quad \cdots\cdots(61)$$

と別表現できる.

以上より，IDCT 値を出力するディジタルフィルタの構成は**図 18.19** のようになり，出力$\{x_{n-3},\ x_{n-2},\ x_{n-1},\ x_n\}$は，

$$\begin{cases} x_n = \phi_0^{(0)}\tilde{C}_n^{(0)} + \phi_0^{(1)}\tilde{C}_n^{(1)} + \phi_0^{(2)}\tilde{C}_n^{(2)} + \phi_0^{(3)}\tilde{C}_n^{(3)} \\ x_{n-1} = \phi_1^{(0)}\tilde{C}_n^{(0)} + \phi_1^{(1)}\tilde{C}_n^{(1)} + \phi_1^{(2)}\tilde{C}_n^{(2)} + \phi_1^{(3)}\tilde{C}_n^{(3)} \\ x_{n-2} = \phi_2^{(0)}\tilde{C}_n^{(0)} + \phi_2^{(1)}\tilde{C}_n^{(1)} + \phi_2^{(2)}\tilde{C}_n^{(2)} + \phi_2^{(3)}\tilde{C}_n^{(3)} \\ x_{n-3} = \phi_3^{(0)}\tilde{C}_n^{(0)} + \phi_3^{(1)}\tilde{C}_n^{(1)} + \phi_3^{(2)}\tilde{C}_n^{(2)} + \phi_3^{(3)}\tilde{C}_n^{(3)} \end{cases} \quad \cdots\cdot(62)$$

であり，DCT 値 $\left\{\tilde{C}_n^{(\ell)}\right\}_{\ell=0}^{\ell=3}$ から元のディジタル信号$\{x_{n-3},\ x_{n-2},\ x_{n-1},\ x_n\}$が得られることがわかる.

たとえば，$t = nT$ [秒]における 4 サンプルのディジタル信号$\{x_{n-3}, x_{n-2}, x_{n-1}, x_n\}$に対する DCT 値が$\left\{\tilde{C}_n^{(0)}, \tilde{C}_n^{(1)}, \tilde{C}_n^{(2)}, \tilde{C}_n^{(3)}\right\}$であり，$n = 3$ に対する出力値は，

$$\begin{cases} x_0 = \phi_0^{(0)}\tilde{C}_3^{(0)} + \phi_0^{(1)}\tilde{C}_3^{(1)} + \phi_0^{(2)}\tilde{C}_3^{(2)} + \phi_0^{(3)}\tilde{C}_3^{(3)} \\ x_1 = \phi_1^{(0)}\tilde{C}_3^{(0)} + \phi_1^{(1)}\tilde{C}_3^{(1)} + \phi_1^{(2)}\tilde{C}_3^{(2)} + \phi_1^{(3)}\tilde{C}_3^{(3)} \\ x_2 = \phi_2^{(0)}\tilde{C}_3^{(0)} + \phi_2^{(1)}\tilde{C}_3^{(1)} + \phi_2^{(2)}\tilde{C}_3^{(2)} + \phi_2^{(3)}\tilde{C}_3^{(3)} \\ x_3 = \phi_3^{(0)}\tilde{C}_3^{(0)} + \phi_3^{(1)}\tilde{C}_3^{(1)} + \phi_3^{(2)}\tilde{C}_3^{(2)} + \phi_3^{(3)}\tilde{C}_3^{(3)} \end{cases} \quad \cdots\cdots(63)$$

であるので，元のディジタル信号が得られることになる．

さらに続けて，$n=4$，5，…と計算を進めていくと，

1）$n=4$のとき

$$\begin{cases} x_1=\phi_0^{(0)}\tilde{C}_4^{(0)}+\phi_0^{(1)}\tilde{C}_4^{(1)}+\phi_0^{(2)}\tilde{C}_4^{(2)}+\phi_0^{(3)}\tilde{C}_4^{(3)} \\ x_2=\phi_1^{(0)}\tilde{C}_4^{(0)}+\phi_1^{(1)}\tilde{C}_4^{(1)}+\phi_1^{(2)}\tilde{C}_4^{(2)}+\phi_1^{(3)}\tilde{C}_4^{(3)} \\ x_3=\phi_2^{(0)}\tilde{C}_4^{(0)}+\phi_2^{(1)}\tilde{C}_4^{(1)}+\phi_2^{(2)}\tilde{C}_4^{(2)}+\phi_2^{(3)}\tilde{C}_4^{(3)} \\ x_4=\phi_3^{(0)}\tilde{C}_4^{(0)}+\phi_3^{(1)}\tilde{C}_4^{(1)}+\phi_3^{(2)}\tilde{C}_4^{(2)}+\phi_3^{(3)}\tilde{C}_4^{(3)} \end{cases} \quad \cdots\cdots(64)$$

2）$n=5$のとき

$$\begin{cases} x_2=\phi_0^{(0)}\tilde{C}_5^{(0)}+\phi_0^{(1)}\tilde{C}_5^{(1)}+\phi_0^{(2)}\tilde{C}_5^{(2)}+\phi_0^{(3)}\tilde{C}_5^{(3)} \\ x_3=\phi_1^{(0)}\tilde{C}_5^{(0)}+\phi_1^{(1)}\tilde{C}_5^{(1)}+\phi_1^{(2)}\tilde{C}_5^{(2)}+\phi_1^{(3)}\tilde{C}_5^{(3)} \\ x_4=\phi_2^{(0)}\tilde{C}_5^{(0)}+\phi_2^{(1)}\tilde{C}_5^{(1)}+\phi_2^{(2)}\tilde{C}_5^{(2)}+\phi_2^{(3)}\tilde{C}_5^{(3)} \\ x_5=\phi_3^{(0)}\tilde{C}_5^{(0)}+\phi_3^{(1)}\tilde{C}_5^{(1)}+\phi_3^{(2)}\tilde{C}_5^{(2)}+\phi_3^{(3)}\tilde{C}_5^{(3)} \end{cases} \quad \cdots\cdots(65)$$

となり，元のディジタル信号がフィルタ出力されることがわかる．ここで，**図18.19**に示すフィルタバンクの各合成DFのタップ係数は，式(53)の直交基底ベクトル$\left\{\boldsymbol{\phi}^{(\ell)}\right\}_{\ell=0}^{\ell=3}$の各要素値$\left\{\boldsymbol{\phi}_k^{(\ell)}\right\}_{\ell=0}^{\ell=3}$に一致するのである．具体的には，以下のようになる．

① x_nを出力する合成DFのタップ係数

$$\left\{\phi_0^{(0)},\phi_0^{(1)},\phi_0^{(2)},\phi_0^{(3)}\right\} =\left\{1,\sqrt{2}\cos\left(\frac{\pi}{8}\right),\sqrt{2}\cos\left(\frac{2\pi}{8}\right),\sqrt{2}\cos\left(\frac{3\pi}{8}\right)\right\}$$

② x_{n-1}を出力する合成DFのタップ係数

$$\left\{\phi_1^{(0)},\phi_1^{(1)},\phi_1^{(2)},\phi_1^{(3)}\right\} =\left\{1,\sqrt{2}\cos\left(\frac{3\pi}{8}\right),\sqrt{2}\cos\left(\frac{6\pi}{8}\right),\sqrt{2}\cos\left(\frac{9\pi}{8}\right)\right\}$$

③ x_{n-2}を出力する合成DFのタップ係数

$$\left\{\phi_2^{(0)},\phi_2^{(1)},\phi_2^{(2)},\phi_2^{(3)}\right\} =\left\{1,\sqrt{2}\cos\left(\frac{5\pi}{8}\right),\sqrt{2}\cos\left(\frac{10\pi}{8}\right),\sqrt{2}\cos\left(\frac{15\pi}{8}\right)\right\}$$

④ x_{n-3}を出力する合成DFのタップ係数

$$\left\{\phi_3^{(0)},\phi_3^{(1)},\phi_3^{(2)},\phi_3^{(3)}\right\} =\left\{1,\sqrt{2}\cos\left(\frac{7\pi}{8}\right),\sqrt{2}\cos\left(\frac{14\pi}{8}\right),\sqrt{2}\cos\left(\frac{21\pi}{8}\right)\right\}$$

である．

DCT，IDCTのフィルタバンク構成

DCTとIDCTは，それぞれ分析フィルタバンク，合成フィルタバンクとして構成されることを説明した．$N=4$サンプルのディジタル信号に対するDCT，IDCTは，4サンプルを1ブロックとして，ブロックごとの一括処理により実現できることが類推される．すなわち，4サンプルごとにDCTとIDCTの処理をすれば十分であり，$n=3$，7，11，…の時点で変換値を算出することになる．

それでは，具体的な変換処理の流れを示すことにしよう．DCTとIDCTのフィルタバンク構成を，それぞれ図18.20，

〔図18.19〕IDCTの合成フィルタバンクの構成($N=4$)

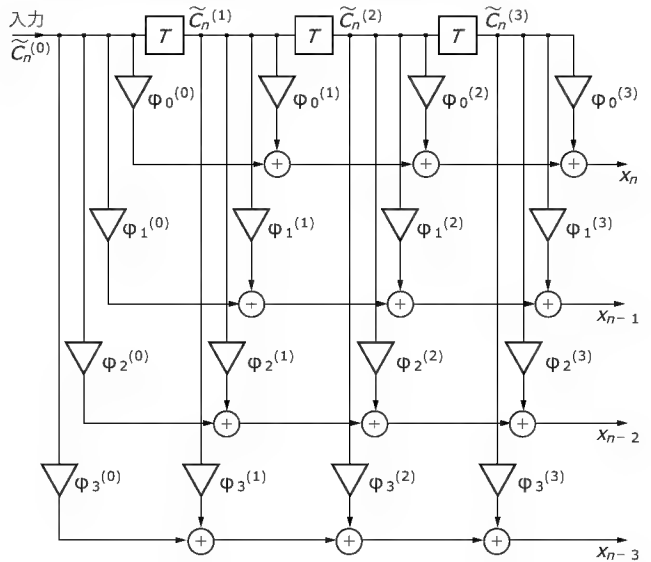



図18.21に示す．**図18.20**と**図18.21**のフィルタバンクに基づくDCTとIDCTは，それぞれ次式で計算することに等価である．

- DCTの計算式

$$\begin{cases} C_0^{(4)}=0.25x_0+0.25x_1+0.25x_2+0.25x_3 \\ C_1^{(4)}=0.3266x_0+0.1353x_1-0.1353x_2-0.3266x_3 \\ C_2^{(4)}=0.25x_0-0.25x_1-0.25x_2+0.25x_3 \\ C_3^{(4)}=0.1353x_0-0.3266x_1+0.3266x_2-0.1353x_3 \end{cases} \cdots(66)$$

- IDCTの計算式

$$\begin{cases} x_0=C_0^{(4)}+1.3065C_1^{(4)}+C_2^{(4)}+0.5411C_3^{(4)} \\ x_1=C_0^{(4)}+0.5411C_1^{(4)}-C_2^{(4)}-1.3065C_3^{(4)} \\ x_2=C_0^{(4)}-0.5411C_1^{(4)}-C_2^{(4)}+1.3065C_3^{(4)} \\ x_3=C_0^{(4)}-1.3065C_1^{(4)}+C_2^{(4)}-0.5411C_3^{(4)} \end{cases} \cdots\cdots(67)$$

いま，入力ディジタル信号系列が，

1，2，3，4，5，5，5，5，4，3，2，1，…

としよう．以下に，式(66)，式(67)によるブロックごとのDCT，IDCTの計算結果を示す．

① 1ブロック目($x_0=1$，$x_1=2$，$x_2=3$，$x_3=4$)

- DCT

$\tilde{C}_3^{(0)}=C_0^{(4)}=2.5$

$\tilde{C}_3^{(1)}=C_1^{(4)}=-1.1151$

$\tilde{C}_3^{(2)}=C_2^{(4)}=0$

$\tilde{C}_3^{(3)}=C_3^{(4)}=-0.0793$

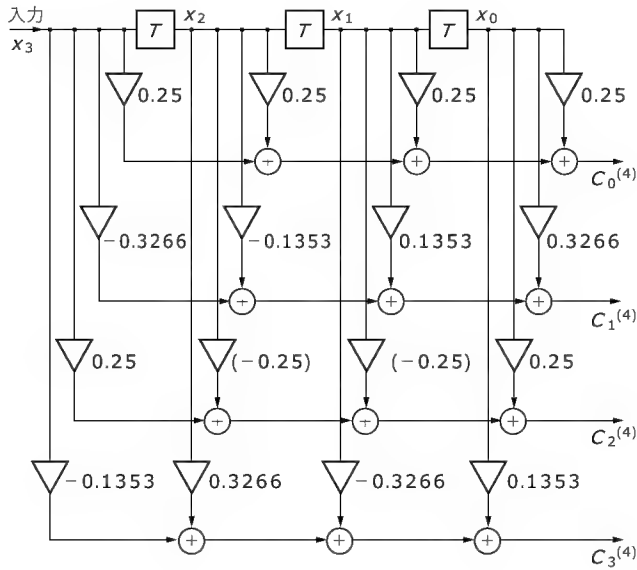
- IDCT

$x_0=1$

$x_1=2$

$x_2=3$

〔図 18.20〕**DCT のフィルタバンク構成例**〔式(66)に基づく〕

〔図 18.21〕**IDCT のフィルタバンク構成例**〔式(67)に基づく〕

$x_3 = 4$

② 2ブロック目 ($\tilde{x}_4 = 5 \rightarrow x_0$, $\tilde{x}_5 = 5 \rightarrow x_1$, $\tilde{x}_6 = 5 \rightarrow x_2$, $\tilde{x}_7 = 5 \rightarrow x_3$)

- DCT

$\tilde{C}_7^{(0)} = C_0^{(4)} = 5$

$\tilde{C}_7^{(1)} = C_1^{(4)} = 0$

$\tilde{C}_7^{(2)} = C_2^{(4)} = 0$

$\tilde{C}_7^{(3)} = C_3^{(4)} = 0$

- IDCT

$\tilde{x}_4 = x_0 = 5$

$\tilde{x}_5 = x_1 = 5$

$\tilde{x}_6 = x_2 = 5$

$\tilde{x}_7 = x_3 = 5$

③ 3ブロック目 ($\tilde{x}_8 = 4 \rightarrow x_0$, $\tilde{x}_9$ $\rightarrow 3$ x_1= $\tilde{x}_{10} \rightarrow 2$ x_2, $\tilde{x}_{11} = 1 \rightarrow x_3$)

- DCT

$\tilde{C}_{11}^{(0)} = C_0^{(4)} = 2.5$

$\tilde{C}_{11}^{(1)} = C_1^{(4)} = 1.1151$

$\tilde{C}_{11}^{(2)} = C_2^{(4)} = 0$

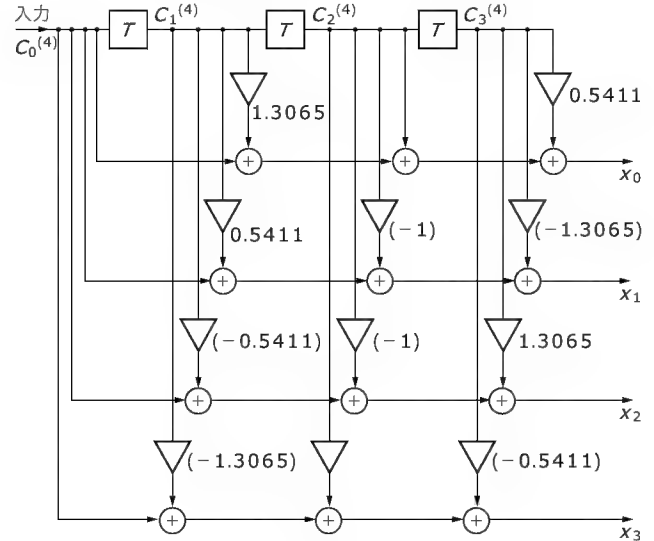
$\tilde{C}_{11}^{(3)} = C_3^{(4)} = 0.0793$

- IDCT

$\tilde{x}_8 = x_0 = 4$

$\tilde{x}_9 = x_1 = 3$

$\tilde{x}_{10} = x_2 = 2$

$\tilde{x}_{11} = x_3 = 1$

以上の結果からわかるように，DCT と IDCT とが相互に逆変換の関係にあり，分析と合成のフィルタバンク構成の妥当性を検証することができる．

最後に DCT と IDCT をフィルタバンク構成からの見方をまとめると，以下のようになる．

DCT ：『オーディオや画像などのディジタル信号を周波数成分ごとに分解するためのフィルタバンク』

IDCT ：『帯域ごとに分解された周波数成分から元のディジタル信号を再合成するためのフィルタバンク』

＊　　　＊

次回は，マルチレート信号処理とよばれる手法に基づき，DCT，IDCT を実現するシステムを採り上げて，フィルタバンクの実用的な構成について解説する予定である．お楽しみに．

みたに・まさあき　東京電機大学工学部情報通信工学科

x86CPUだけでもマスタしたい

開発技術者のためのアセンブラ入門

第21回 FPU命令の概略

大貫広幸

今回は，x86系の32ビットCPUがもつ浮動小数点演算命令について，その概略とアセンブラMASMおよびgasでの使用法について説明します．

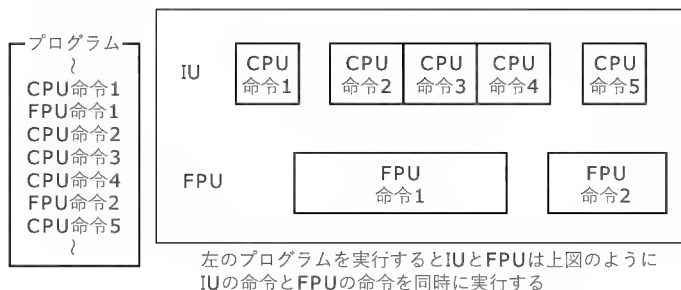
CPUとFPUの関係

浮動小数点演算は，浮動小数点演算ユニット(FPU)が行います．FPUは，CPU本体がもつ整数演算ユニット(IU)とは独立して存在するため，x86系の32ビットCPUは，浮動小数点演算と整数演算を同時に実行することが可能です．つまり，プロセッサ内ではFPUとIUは並行処理されるわけです(**図1**)．

これまで説明してきた汎用命令の整数演算は，すべてIUで処理されていました．しかし，独立したユニットであるFPUには，FPU専用の「FPU命令」と呼ばれる浮動小数点演算命令が使われます．FPU命令は，正式にはインテルのマニュアル(IA-32インテルアーキテクチャソフトウェアデベロッパーズマニュアル)によると「x87 FPU命令」となっています．

このx87とは，昔のx86系CPU(8086や8088，80286)本体にはなかった，浮動小数点演算を行うためのコプロセッサに付けられた名称からきています．Pentium以降のプロセッサはすべて，チップ内部に浮動小数点演算ユニット(FPU)を内蔵しているので，このコプロセッサは現在使用されていません(**図2**)．そのため，現在のPentium4でも，多少命令は追加されていますが，基本的な浮動小数点演算の機能は，この初期のx87コプロセッサとあまり変わりありません．

〔図1〕IUとFPU

左のプログラムを実行するとIUとFPUは上図のように IUの命令とFPUの命令を同時に実行する

FPU命令が扱うデータ

FPU命令では，連載第3回(2001年2月号)で説明した2進浮動小数点で表される「単精度(4バイト長)」，「倍精度(8バイト長)」，「拡張精度(10バイト長)」の三つの実数と，符号付き2進整数の「ワード(2バイト長)」，「ショート(4バイト長)」，「ロング(8バイト長)」，そしてパックドBCD(10バイト長)の計七つのデータを扱うことができます．これらのデータは，メモリ上での記憶形式で，FPU上ではすべてがいちばん精度が高く絶対値が大きい拡張精度で記憶，演算されます．

アセンブラMASMおよびgasでは，これらデータを定義するため，**表1**のようなディレクティブを使用します．

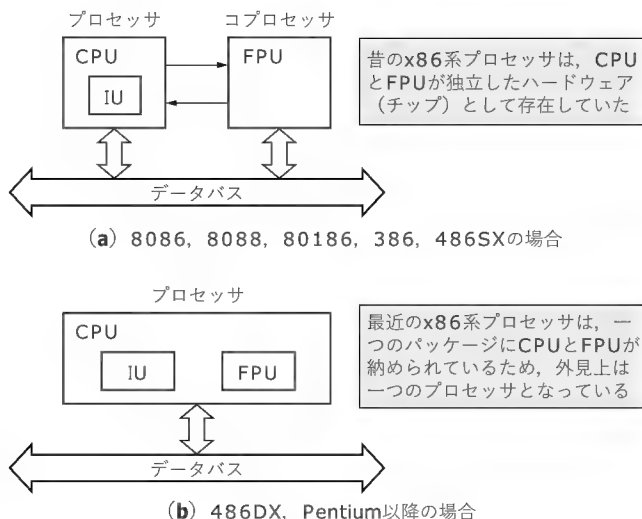
● MASMの場合

MASMでは，単精度は「REAL4」，倍精度は「REAL8」，そして拡張精度は「REAL10」のディレクティブで定義します．

REAL4，REAL8，REAL10の構文は，整数を定義するDBやDWと同じです．ただし，初期値を指定する場合，値として小数点をもった実数を記述する必要があります．もし，初期値に指数が必要であれば，小数点をもった実数のすぐ右に，文字E

〔図2〕CPUとFPU

(a) 8086，8088，80186，386，486SXの場合

(b) 486DX，Pentium以降の場合

〔表1〕FPU命令で使用するデータを定義するディレクティブ

データの種類		MASMのディレクティブ	gasのディレクティブ
実数	単精度	REAL4	.float .single
	倍精度	REAL8	.double
	拡張精度	REAL10	.tfloat
符号付き2進整数	ワード	SWORD	.word .short .hword
	ショート	SDWORD	.long .int
	ロング	DQ QWORD	.quad
パックドBCD		DT TBYTE	なし

と10のn乗に対応するnを書きます．

実際のディレクティブREAL4，REAL8，REAL10の記述例を，**リスト1**に示します．

符号付き2進整数のワード，ショート，ロングは，2進整数を定義するディレクティブを使用します．

パックドBCDは，TBYTEのディレクティブを使用しますが，BCDで表された定数を初期値として設定することはできません．どうしても，BCDで表された定数を初期値として設定したい場合は，16進数の表記で指定します．

● gasの場合

gasでは，初期値をもったデータは，単精度は「.float」あるいは「.single」，倍精度は「.double」，そして拡張精度は

〔リスト1〕MASMのFPU命令の記述例

```
                              .586
                              .model flat

00000000                      .data

00000000 3F9D70A4             f4      real4     1.23,-1.23
          BF9D70A4
00000008                      f8      real8     1.23e10,-1.23e10
          4206E918D8000000
          C206E918D8000000
00000018                      f10     real10    1.23e-10,-1.23e-10
          3FDE873D6C1110F76374
          BFDE873D6C1110F76374

0000002C 0002                 i2      sword   2
0000002E 00000004             i4      sdword  4
00000032                      i8      qword   8
          0000000000000008

0000003A                      d10     tbyte     123456h
          00000000000000123456

00000000                      .code
00000000  D9 C3                       fld     st(3)
00000002  D9 05 00000000 R            fld     f4
00000008  DD 05 00000008 R            fld     f8
0000000E  DB 2D 00000018 R            fld     f10
00000014  DF 05 0000002C R            fild    i2
0000001A  DB 05 0000002E R            fild    i4
00000020  DF 2D 00000032 R            fild    i8
00000026  DF 25 0000003A R            fbld    d10

0000002C  DD D3                       fst     st(3)
0000002E  D9 15 00000000 R            fst     f4
00000034  DD 15 00000008 R            fst     f8
0000003A  DF 15 0000002C R            fist    i2
00000040  DB 15 0000002E R            fist    i4

00000046  DD DB                       fstp    st(3)
00000048  D9 1D 00000000 R            fstp    f4
0000004E  DD 1D 00000008 R            fstp    f8
00000054  DB 3D 00000018 R            fstp    f10
0000005A  DF 1D 0000002C R            fistp   i2
00000060  DB 1D 0000002E R            fistp   i4
00000066  DF 3D 00000032 R            fistp   i8
0000006C  DF 35 0000003A R            fbstp   d10

00000072  D9 C9                       fxch    st(1)
00000074  D9 C9                       fxch

00000076  D8 05 00000000 R            fadd    f4             ; ST(0) <- ST(0)+f4
0000007C  DC 05 00000008 R            fadd    f8             ; ST(0) <- ST(0)+f8
00000082  DE 05 0000002C R            fiadd   i2             ; ST(0) <- ST(0)+i2
00000088  DA 05 0000002E R            fiadd   i4             ; ST(0) <- ST(0)+i4
0000008E  D8 C1                       fadd    st,st(1)       ; ST(0) <- ST(0)+ST(1)
00000090  DC C1                       fadd    st(1),st       ; ST(1) <- ST(1)+ST(0)
00000092  DE C1                       faddp   st(1),st       ; ST(1) <- ST(1)+ST(0); pop
00000094  DE C1                       fadd                   ; ST(1) <- ST(1)+ST(0); pop
```

「.tfloat」のディレクティブで定義します．

.float，.single，.double，.tfloatの構文は，整数を定義する.byteや.wordと同じです．ただし，初期値は小数点をもった実数で記述する必要があります．もし，初期値に指数が必要であれば，小数点をもった実数のすぐ右に，文字Eと10のn乗に対応するnを書きます．

実際のディレクティブ.float，.single，.double，.tfloatの記述例を**リスト2**に示します．符号付き2進整数のワード，ショート，ロングは，2進整数を定義するディレクティブを使用します．gasにはパックドBCDのための10バイトデータの定義はありません．そのため，初期値をもったパックドBCDのデータは定義できません．

FPUのレジスタ

FPUのレジスタは**図3**(p.130)のような構成をしています．各レジスタは，次のような役割をもっています．

- **レジスタスタック(80ビット長×8本)**

演算に使用する拡張精度(80ビット長)の値を格納しておくためのレジスタが，レジスタスタックと呼ばれる8本のレジスタです．レジスタスタックは，その名のようにスタック構造をしています(**図4**，p.131)．

〔リスト1〕MASMのFPU命令の記述例(つづき)

```
00000096  D8 25 00000000 R            fsub    f4            ; ST(0) <- ST(0)-f4
0000009C  DC 25 00000008 R            fsub    f8            ; ST(0) <- ST(0)-f8
000000A2  DE 25 0000002C R            fisub   i2            ; ST(0) <- ST(0)-i2
000000A8  DA 25 0000002E R            fisub   i4            ; ST(0) <- ST(0)-i4
000000AE  D8 E1                       fsub    st,st(1)      ; ST(0) <- ST(0)-ST(1)
000000B0  DC E9                       fsub    st(1),st      ; ST(1) <- ST(1)-ST(0)
000000B2  DE E9                       fsubp   st(1),st      ; ST(1) <- ST(1)-ST(0); pop
000000B4  DE E9                       fsub                  ; ST(1) <- ST(1)-ST(0); pop

000000B6  D8 2D 00000000 R            fsubr   f4            ; ST(0) <- f4-ST(0)
000000BC  DC 2D 00000008 R            fsubr   f8            ; ST(0) <- f8-ST(0)
000000C2  DE 2D 0000002C R            fisubr  i2            ; ST(0) <- i2-ST(0)
000000C8  DA 2D 0000002E R            fisubr  i4            ; ST(0) <- i4-ST(0)
000000CE  D8 E9                       fsubr   st,st(1)      ; ST(0) <- ST(1)-ST(0)
000000D0  DC E1                       fsubr   st(1),st      ; ST(1) <- ST(0)-ST(1)
000000D2  DE E1                       fsubrp  st(1),st      ; ST(1) <- ST(0)-ST(1); pop
000000D4  DE E1                       fsubr                 ; ST(1) <- ST(0)-ST(1); pop

000000D6  D8 0D 00000000 R            fmul    f4            ; ST(0) <- ST(0)×f4
000000DC  DC 0D 00000008 R            fmul    f8            ; ST(0) <- ST(0)×f8
000000E2  DE 0D 0000002C R            fimul   i2            ; ST(0) <- ST(0)×i2
000000E8  DA 0D 0000002E R            fimul   i4            ; ST(0) <- ST(0)×i4
000000EE  D8 C9                       fmul    st,st(1)      ; ST(0) <- ST(0)×ST(1)
000000F0  DC C9                       fmul    st(1),st      ; ST(1) <- ST(1)×ST(0)
000000F2  DE C9                       fmulp   st(1),st      ; ST(1) <- ST(1)×ST(0); pop
000000F4  DE C9                       fmul                  ; ST(1) <- ST(1)×ST(0); pop

000000F6  D8 35 00000000 R            fdiv    f4            ; ST(0) <- ST(0)÷f4
000000FC  DC 35 00000008 R            fdiv    f8            ; ST(0) <- ST(0)÷f8
00000102  DE 35 0000002C R            fidiv   i2            ; ST(0) <- ST(0)÷i2
00000108  DA 35 0000002E R            fidiv   i4            ; ST(0) <- ST(0)÷i4
0000010E  D8 F1                       fdiv    st,st(1)      ; ST(0) <- ST(0)÷ST(1)
00000110  DC F9                       fdiv    st(1),st      ; ST(1) <- ST(1)÷ST(0)
00000112  DE F9                       fdivp   st(1),st      ; ST(1) <- ST(1)÷ST(0); pop
00000114  DE F9                       fdiv                  ; ST(1) <- ST(1)÷ST(0); pop

00000116  D8 3D 00000000 R            fdivr   f4            ; ST(0) <- f4÷ST(0)
0000011C  DC 3D 00000008 R            fdivr   f8            ; ST(0) <- f8÷ST(0)
00000122  DE 3D 0000002C R            fidivr  i2            ; ST(0) <- i2÷ST(0)
00000128  DA 3D 0000002E R            fidivr  i4            ; ST(0) <- i4÷ST(0)
0000012E  D8 F9                       fdivr   st,st(1)      ; ST(0) <- ST(1)÷ST(0)
00000130  DC F1                       fdivr   st(1),st      ; ST(1) <- ST(0)÷ST(1)
00000132  DE F1                       fdivrp  st(1),st      ; ST(1) <- ST(0)÷ST(1); pop
00000134  DE F1                       fdivr                 ; ST(1) <- ST(0)÷ST(1); pop

00000136  D8 D1                       fcom    st(1)         ; ST(0) 比較 ST(1)
00000138  D8 D1                       fcom                  ; ST(0) 比較 ST(1)
0000013A  9B DF E0                    fstsw   ax
0000013D  9E                          sahf
0000013E  74 02                       je      skip1

00000140  D9 FA                       fsqrt
00000142                      skip1:
00000142  D9 F2                       fptan
00000144  D9 F3                       fpatan
00000146  D9 F0                       f2xm1
00000148  D9 F1                       fyl2x
0000014A  D9 F9                       fyl2xp1

0000014C  9B                          fwait
0000014D  9B                          wait
                              end
```

オペランドに演算順序があるSUBとDIVはMASMとgasでオペランドの指定と演算順序が異なるので注意

FPUの比較結果は，このようにしてCPUのフラグにセットする

〔リスト2〕gasのFPU命令の記述例

```
 1                    .data
 2  0000 A4709D3F     f4:     .float     1.23,-1.23
 2       A4709DBF
 3  0008 000000D8     f8:     .double    1.23e10,-1.23e10
 3       18E90642
 3       000000D8
 3       18E906C2
 4  0018 7463F710     f10:    .tfloat    1.23e-10,-1.23e-10
 4       116C3D87
 4       DE3F7463
 4       F710116C
 4       3D87DEBF
 5
 6  002c 0200         i2:     .word   2
 7  002e 04000000     i4:     .long   4
 8  0032 08000000     i8:     .quad   8
 8       00000000
 9
10  003a 00000000     d10:    .space  10
10       00000000
10       0000
11
12  0044 A4709D3F             .single    1.23,-1.23
12       A4709DBF
13
14                    .text
15  0000 D9C3                 fld     %st(3)
16  0002 D9050000             flds    f4
16       0000
17  0008 DD050800             fldl    f8
17       0000
18  000e DB2D1800             fldt    f10
18       0000
19  0014 DF052C00             filds   i2
19       0000
20  001a DB052E00             fildl   i4
20       0000
21  0020 DF2D3200             fildq   i8
21       0000
22  0026 DF253A00             fbld    d10
22       0000
23
24  002c DDD3                 fst     %st(3)
25  002e D9150000             fsts    f4
25       0000
26  0034 DD150800             fstl    f8
26       0000
27  003a DF152C00             fists   i2
27       0000
28  0040 DB152E00             fistl   i4
28       0000
29
30  0046 DDDB                 fstp    %st(3)
31  0048 D91D0000             fstps   f4
31       0000
32  004e DD1D0800             fstpl   f8
32       0000
33  0054 DB3D1800             fstpt   f10
33       0000
34  005a DF1D2C00             fistps  i2
34       0000
35  0060 DB1D2E00             fistpl  i4
35       0000
36  0066 DF3D3200             fistpq  i8
36       0000
37  006c DF353A00             fbstp   d10
37       0000
38
39  0072 D9C9                 fxch    %st(1)
40  0074 D9C9                 fxch
41
42  0076 D8050000             fadds   f4              # ST(0)+f4 -> ST(0)
42       0000
43  007c DC050800             faddl   f8              # ST(0)+f8 -> ST(0)
43       0000
44  0082 DE052C00             fiadds  i2              # ST(0)+i2 -> ST(0)
44       0000
45  0088 DA052E00             fiaddl  i4              # ST(0)+i4 -> ST(0)
45       0000
46  008e D8C1                 fadd    %st(1),%st      # ST(0)+ST(1) -> ST(0)
47  0090 DCC1                 fadd    %st,%st(1)      # ST(0)+ST(1) -> ST(1)
48  0092 DEC1                 faddp   %st,%st(1)      # ST(0)+ST(1) -> ST(1); pop
49  0094 DEC1                 faddp                   # ST(0)+ST(1) -> ST(1); pop
```

浮動小数点データ(実数)は，単精度が.floatあるいは.single，倍精度が.double，拡張精度が.tfloatのディレクティブで定義する

FPUのレジスタスタックは %st(i) と指定する

〔リスト2〕gas の FPU 命令の記述例(つづき)

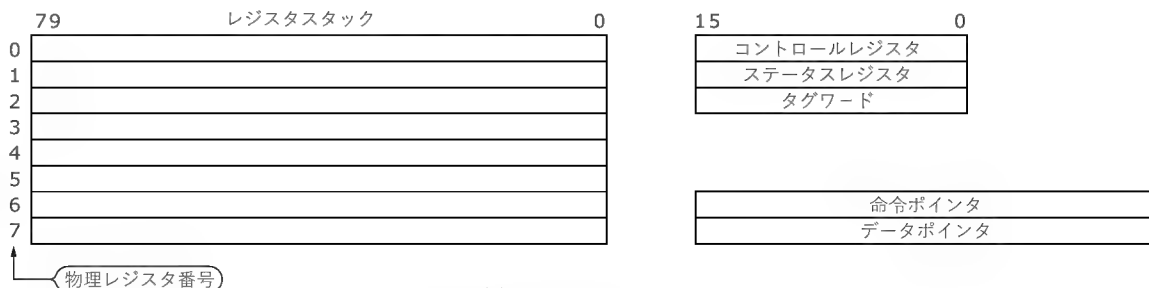
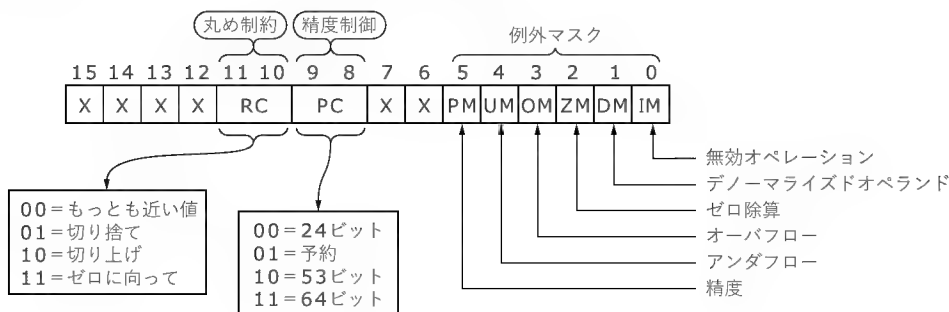
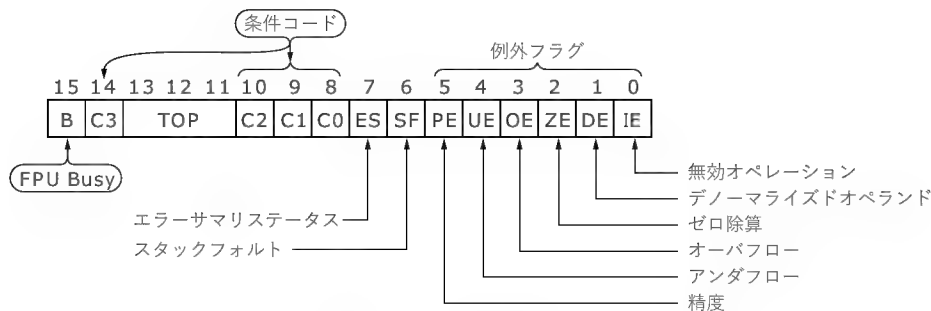
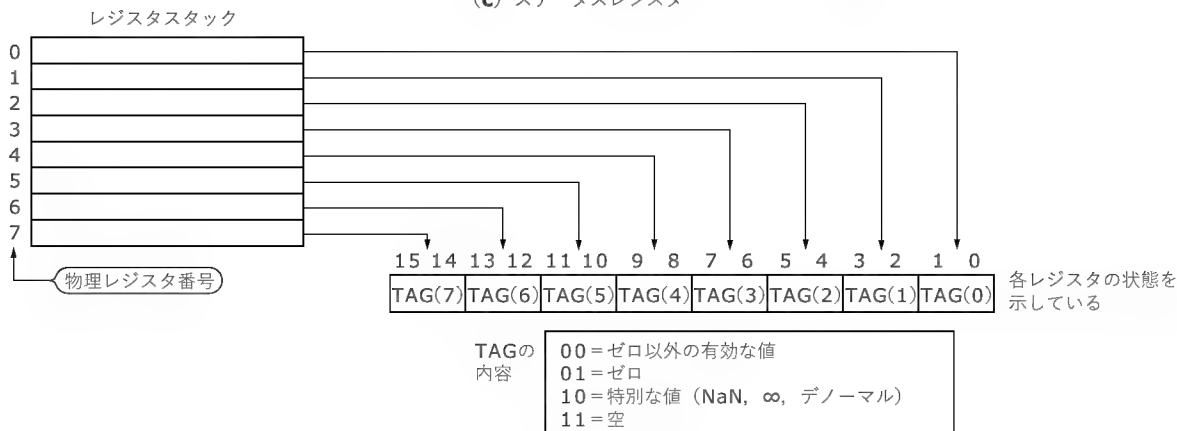
```
 50
 51  0096 D8250000              fsubs   f4            # ST(0)-f4 -> ST(0)
 51       0000
 52  009c DC250800              fsubl   f8            # ST(0)-f8 -> ST(0)
 52       0000
 53  00a2 DE252C00              fisubs  i2            # ST(0)-i2 -> ST(0)
 53       0000
 54  00a8 DA252E00              fisubl  i4            # ST(0)-i4 -> ST(0)
 54       0000
 55  00ae D8E1                  fsub    %st(1),%st    # ST(0)-ST(1) -> ST(0)
 56  00b0 DCE1                  fsub    %st,%st(1)    # ST(0)-ST(1) -> ST(1)
 57  00b2 DEE1                  fsubp   %st,%st(1)    # ST(0)-ST(1) -> ST(1); pop
 58  00b4 DEE1                  fsubp                 # ST(0)-ST(1) -> ST(1); pop
 59
 60  00b6 D82D0000              fsubrs  f4            # f4-ST(0) -> ST(0)
 60       0000
 61  00bc DC2D0800              fsubrl  f8            # f8-ST(0) -> ST(0)
 61       0000
 62  00c2 DE2D2C00              fisubrs i2            # i2-ST(0) -> ST(0)
 62       0000
 63  00c8 DA2D2E00              fisubrl i4            # i4-ST(0) -> ST(0)
 63       0000
 64  00ce D8E9                  fsubr   %st(1),%st    # ST(1)-ST(0) -> ST(0)
 65  00d0 DCE9                  fsubr   %st,%st(1)    # ST(1)-ST(0) -> ST(1)
 66  00d2 DEE9                  fsubrp  %st,%st(1)    # ST(1)-ST(0) -> ST(1); pop
 67  00d4 DEE9                  fsubrp                # ST(1)-ST(0) -> ST(1); pop
 68
 69  00d6 D80D0000              fmuls   f4            # ST(0)×f4 -> ST(0)
 69       0000
 70  00dc DC0D0800              fmull   f8            # ST(0)×f8 -> ST(0)
 70       0000
 71  00e2 DE0D2C00              fimuls  i2            # ST(0)×i2 -> ST(0)
 71       0000
 72  00e8 DA0D2E00              fimull  i4            # ST(0)×i4 -> ST(0)
 72       0000
 73  00ee D8C9                  fmul    %st(1),%st    # ST(0)×ST(1) -> ST(0)
 74  00f0 DCC9                  fmul    %st,%st(1)    # ST(0)×ST(1) -> ST(1)
 75  00f2 DEC9                  fmulp   %st,%st(1)    # ST(0)×ST(1) -> ST(1); pop
 76  00f4 DEC9                  fmulp                 # ST(0)×ST(1) -> ST(1); pop
 77
 78  00f6 D8350000              fdivs   f4            # ST(0)÷f4 -> ST(0)
 78       0000
 79  00fc DC350800              fdivl   f8            # ST(0)÷f8 -> ST(0)
 79       0000
 80  0102 DE352C00              fidivs  i2            # ST(0)÷i2 -> ST(0)
 80       0000
 81  0108 DA352E00              fidivl  i4            # ST(0)÷i4 -> ST(0)
 81       0000
 82  010e D8F1                  fdiv    %st(1),%st    # ST(0)÷ST(1) -> ST(0)
 83  0110 DCF1                  fdiv    %st,%st(1)    # ST(0)÷ST(1) -> ST(1)
 84  0112 DEF1                  fdivp   %st,%st(1)    # ST(0)÷ST(1) -> ST(1); pop
 85  0114 DEF1                  fdivp                 # ST(0)÷ST(1) -> ST(1); pop
 86
 87  0116 D83D0000              fdivrs  f4            # f4÷ST(0) -> ST(0)
 87       0000
 88  011c DC3D0800              fdivrl  f8            # f8÷ST(0) -> ST(0)
 88       0000
 89  0122 DE3D2C00              fidivrs i2            # i2÷ST(0) -> ST(0)
 89       0000
 90  0128 DA3D2E00              fidivrl i4            # i4÷ST(0) -> ST(0)
 90       0000
 91  012e D8F9                  fdivr   %st(1),%st    # ST(1)÷ST(0) -> ST(0)
 92  0130 DCF9                  fdivr   %st,%st(1)    # ST(1)÷ST(0) -> ST(1)
 93  0132 DEF9                  fdivrp  %st,%st(1)    # ST(1)÷ST(0) -> ST(1); pop
 94  0134 DEF9                  fdivrp                # ST(1)÷ST(0) -> ST(1); pop
 95
 96  0136 D8D1                  fcom    %st(1)        # ST(0) 比較 ST(1)
 97  0138 D8D1                  fcom                  # ST(0) 比較 ST(1)
 98  013a 9BDFE0                fstsw   %ax
 99  013d 9E                    sahf
100  013e 7402                  jz      skip1
101
102  0140 D9FA                  fsqrt
103                     skip1:
104  0142 D9F2                  fptan
105  0144 D9F3                  fpatan
106  0146 D9F0                  f2xm1
107  0148 D9F1                  fyl2x
108  014a D9F9                  fyl2xp1
109
110  014c 9B                    fwait
111  014d 9B                    wait
```

SUBとDIVの命令はgasのオペランド指定と演算順序がMASMと異なるので注意が必要

値のロードでは，値はスタックトップにPUSHされ，その値が新しいスタックトップとなります．逆に値のストアでは，スタックトップの値が使われます．演算もスタックトップを中心に行われます．

スタックトップの値は，命令によっては動作終了後にPOPし，スタックトップの値を捨てることもできます．この場合，POP前のスタックトップの次の値が，POP後，新しいスタックトップとなります．

レジスタスタック上の各レジスタには，物理レジスタ番号の他に，常にスタックトップを0とするような相対的なレジスタ

〔図3〕
FPUのレジスタ構成

(a) FPUの全レジスタ

(b) コントロールレジスタ

(c) ステータスレジスタ

(d) タグワード

番号が付けられています．FPU命令では，この相対的なレジスタ番号をオペランドとして指定します．

FPU命令でスタックトップのレジスタをSTあるいはST(0)と表し，スタックトップのすぐ下のレジスタをST(1)と，そしてその下がST(2)，ST(3)，…，ST(7)と表します．アセンブラMASMでは，ST，ST(0)，ST(1)，…，ST(7)がレジスタ名として使われます．gasではさらに%を付けて%st，%st(0)，%st(1)，…，%st(7)がレジスタ名として使われます．

図5は，レジスタスタックを使ってどのように演算するかを示した例です．

● タグワード(16ビット長)

1フィールドが2ビットのTAG(0)～TAG(7)で，レジスタスタック上の各レジスタの状態を示しています．TAG(0)～TAG(7)は，レジスタスタックの物理レジスタ番号0～7に対応しています．状態は，「ゼロ以外の有効な値」，「ゼロ」，「特別な値」，「空」の4種類を表します．

● コントロールレジスタとコントロールワード(16ビット長)

FPUの制御するためのレジスタで，このレジスタに設定する値のことをコントロールワードと呼びます．コントロールワードでは，丸め制御や精度制御，例外マスクを指定します．

(1) 丸め制御

演算結果の格納やメモリへの値のストアで，高い精度の値を低い精度の値に変換するとき，失われてしまう有効数字部の下位ビットの処理方法を指定するものです．丸め制御では，「もっとも近い値」，「切り捨て(－∞方向)」，「切り上げ(＋∞方向)」，「ゼロに向かって」の四つの指定ができます．初期値では丸めはもっとも近い値になっています．

(2) 精度制御

演算時の精度を指定するもので，プログラムを移植するとき，移植対象のプログラムが使用していた演算プロセッサや演算プログラムと互換を取るために使用されます．精度では，演算時の有効数字部のビットを単精度相当の「24ビット」，倍精度相当の「53ビット」，拡張精度時の「64ビット」の3種類から選択できます．初期値ではいちばん精度が高い有効数字部64ビットになっています．

(3) 例外マスク

例外マスクのビットには，「精度」，「アンダフロー」，「オーバフロー」，「ゼロ除算」，「デノーマライズドオペランド」，「無

〔図4〕FPUのレジスタスタックの動作

〔図5〕FPUのレジスタスタックでの演算例

〔表2〕FPU命令の一覧

分　類	インストラクション名	動　作
データ転送命令	FLD	浮動小数点値のロード，値はPUSHされST(0)となる
	FST	ST(0)の浮動小数点値のストア
	FSTP	ST(0)の浮動小数点値をストア後，POPし元のST(0)の値を捨てる
	FILD	整数値のロード，値はPUSHされST(0)となる
	FIST	ST(0)を整数値としてストア
	FISTP	ST(0)を整数値としてストア後，POPし元のST(0)の値を捨てる
	FBLD	パックドBCDの値をロード，値はPUSHされST(0)となる
	FBSTP	ST(0)をパックドBCDとしてストア後，POPし元のST(0)の値を捨てる
	FXCH	ST(0)とST(i)の値の交換
	FCMOVcc*	ccの条件成立で，ST(i)をST(0)に転送 cc=(E，NE，B，BE，NB，NBE，U，NU)
基本演算命令	FADD	浮動小数点加算
	FADDP	浮動小数点加算後，POPし元のST(0)の値を捨てる
	FIADD	整数値を浮動小数点に変換後，浮動小数点加算
	FSUB	浮動小数点減算
	FSUBP	浮動小数点減算後，POPし元のST(0)の値を捨てる
	FISUB	整数値を浮動小数点に変換後，浮動小数点減算
	FSUBR	オペランドを逆にして浮動小数点減算
	FSUBRP	オペランドを逆にして浮動小数点減算後，POPし元のST(0)の値を捨てる
	FISUBR	整数値を浮動小数点に変換後，オペランドを逆にして浮動小数点減算
	FMUL	浮動小数点乗算
	FMULP	浮動小数点乗算後，POPし元のST(0)の値を捨てる
	FIMUL	整数値を浮動小数点に変換後，浮動小数点乗算
	FDIV	浮動小数点除算
	FDIVP	浮動小数点除算後，POPし元のST(0)の値を捨てる
	FIDIV	整数値を浮動小数点に変換後，浮動小数点除算
	FDIVR	オペランドを逆にして浮動小数点除算
	FDIVRP	オペランドを逆にして浮動小数点除算後，POPし元のST(0)の値を捨てる
	FIDIVR	整数値を浮動小数点に変換後，オペランドを逆にして浮動小数点除算
	FPREM	浮動小数点で剰余を求める(x87 FPU独自の演算)
	FPREM1	浮動小数点で剰余を求める(IEEE規格互換の演算)
	FABS	ST(0)の絶対値
	FCHS	ST(0)の符号反転
	FRNDINT	ST(0)の整数への丸め
	FSCALE	スケール(ST(0) ← ST(0) × $2^{ST(1)}$)
	FSQRT	ST(0)の平方根
	FXTRACT	ST(0)の指数部と有効数字部を分解し，元のST(0)に指数，有効数字はPUSHされ新しいST(0)になる
比較命令	FCOM	浮動小数点比較
	FCOMP	浮動小数点比較後，POPし元のST(0)の値を捨てる
	FCOMPP	ST(0)とST(1)を浮動小数点比較後，2回POPし元のST(0)とST(1)の値を捨てる
	FUCOM	順序化不可能比較
	FUCOMP	順序化不可能比較後，POPし元のST(0)の値を捨てる
	FUCOMPP	ST(0)とST(1)を順序化不可能比較後，2回POPし元のST(0)とST(1)の値を捨てる
	FICOM	整数値を浮動小数点に変換後，浮動小数点比較
	FICOMP	整数値を浮動小数点に変換後，浮動小数点比較
比較命令	FICOMP	その後POPし元のST(0)の値を捨てる
	FCOMI*	浮動小数点比較し，その結果をCPUのEFLAGSに設定
	FUCOMI*	順序化不可能比較し，その結果をCPUのEFLAGSに設定
	FCOMIP*	浮動小数点比較し，その結果をCPUのEFLAGSに設定． その後，POPし元のST(0)の値を捨てる
	FUCOMIP*	順序化不可能比較し，その結果をCPUのEFLAGSに設定． その後，POPし元のST(0)の値を捨てる
	FTST	ST(0)と0.0を比較
	FXAM	ST(0)の状態を調べる
超越関数命令	FSIN	ST(0)の正弦
	FCOS	ST(0)の余弦
	FSINCOS	ST(0)の正弦と余弦を同時に計算する．元のST(0)に正弦，余弦はPUSHされ新しいST(0)になる
	FPTAN	ST(0)の正接を計算．元のST(0)に正接，その後1.0がPUSHされ新しいST(0)になる
	FPATAN	ST(1)/ST(0)の結果で逆正接を計算し，ST(1)にストア． その後，POPしST(0)の値は捨てられ，逆正接の値が新しいST(0)になる
	F2XM1	$2^{ST(0)}-1$を計算
	FYL2X	ST(1) × $\log_2$ST(0)を計算し，ST(1)にストア． その後，POPしST(0)の値を捨てられ，対数の計算値が新しいST(0)になる
	FYL2XP1	ST(1) × $\log_2$(ST(0)+1)を計算し，ST(1)にストア． その後，POPしST(0)の値を捨てられ，対数の計算値が新しいST(0)になる
定数ロード命令	FLD1	定数＋1.0をST(0)にPUSH
	FLDZ	定数＋0.0をST(0)にPUSH
	FLDPI	定数πをST(0)にPUSH
	FLDL2E	定数$\log_2 e$をST(0)にPUSH
	FLDLN2	定数$\log_e 2$をST(0)にPUSH
	FLDL2T	定数$\log_2 10$をST(0)にPUSH
	FLDLG2	定数$\log_{10} 2$をST(0)にPUSH
x87 FPU制御命令	FINCSTP	FPUのステータスレジスタのTOPを＋1する
	FDECSTP	FPUのステータスレジスタのTOPを－1する
	FFREE	指定ST(i)を空にする
	FINIT	未処理の例外の有無を調べた後，FPUを初期化する
	FNINIT	未処理の例外の有無を調べずに，FPUを初期化する
	FCLEX	未処理の例外の有無を調べた後，例外フラグをクリアする
	FNCLEX	未処理の例外の有無を調べずに，例外フラグをクリアする
	FSTCW	未処理の例外の有無を調べた後，FPUのコントロールワードをストア
	FNSTCW	未処理の例外の有無を調べずに，FPUのコントロールワードをストア
	FLDCW	FPUのコントロールワードのロード
	FSTENV	未処理の例外の有無を調べた後，FPUの環境をストア
	FNSTENV	未処理の例外の有無を調べずに，FPUの環境をストア
	FLDENV	FPUの環境をロード
	FSAVE	未処理の例外の有無を調べた後，FPUの状態をストア

〔表2〕FPU命令の一覧（つづき）

分　類	インストラクション名	動　作
x87 FPU 制御命令	FNSAVE	未処理の例外の有無を調べずに，FPUの状態をストア
	FRSTOR	FPUの状態をロード
	FSTSW	未処理の例外の有無を調べた後，FPUのステータスワードをストア
	FNSTSW	未処理の例外の有無を調べずに，FPUのステータスワードをストア
	WAIT FWAIT	未処理のマスクされていない浮動小数点例外の有無を調べ，あればその処理を実行する
	FNOP	FPUは何もしない

●表中の＊が付けられた命令は，Pentium Pro以降のFPUでのみ使用可能な命令
●表中の「ステータスワード」はステータスレジスタの値，「コントロールワード」はコントロールレジスタの値，「環境」はコントロール，ステータス，タグワード，命令ポインタ，データポインタを指す，「状態」は環境にレジスタスタックを加えたFPUのすべてのレジスタ

効オペレーション」があります．この内，精度は演算結果や値をメモリにストアしたとき丸めが発生したことを示し，デノーマライズドオペランドは，値としてデノーマルな値が使われたことを示します．

例外マスクの各ビットは，例外が発生した場合に，割り込みを発生させるか否かを指定するものです．ビットを0にすることで割り込みが発生します．例外マスクが1の場合に，例外が発生すると，演算結果が例外発生を示す値となります．たとえば，オーバフローやゼロ除算を行った場合なら，無限大が演算結果となります．初期値ではすべてマスク(1)になっています．

● ステータスレジスタとステータスワード（16ビット長）

FPUの状態を表しているレジスタで，このレジスタから得られた値のことをステータスワードと呼びます．ステータスワードでは，FPUの動作中を示す「FPU Busy」，実行結果を表す「条件コード」，レジスタスタックのST(0)の物理レジスタ番号を示す「TOP」，そして例外発生を示す「エラーサマリステータス」，「スタックフォルト」，「例外フラグ」があります．

(1) FPU Busy

古い16ビット時代のFPUとの互換のために用意されているビットです．32ビットCPUでは，エラーサマリステータスと同じ内容となっています．

(2) 条件コード

条件コードは，FPU命令の実行結果が設定されます．ただし，C0～C3の設定のされ方は，命令により異なるので，実際にこの条件コードを使用する場合には注意する必要があります．

(3) 例外フラグとエラーサマリステータス

例外が発生すると，該当する例外フラグが1になります．例外フラグには，例外マスクに対応する形で「精度」，「アンダフロー」，「オーバフロー」，「ゼロ除算」，「デノーマライズドオペランド」，「無効オペレーション」があります．エラーサマリステータスは，例外マスクのいずれかのマスクが0になっているとき，該当する例外が発生し，FPU例外処理ハンドラを呼び出すための割り込みを要求している場合に1になります．

(4) スタックフォルト

レジスタスタックで値のPUSH/POPが行われ，レジスタスタックがオーバフローあるいはアンダフローしたことを示すものです．このとき条件コードのC1が1ならオーバフロー，0ならアンダフローということになります．FPUの例外処理ハンドラに，このスタックフォルトの発生に対応した，レジスタスタック拡張プログラムを組み込むことで，レジスタスタックをメモリ上に拡張することができます．

これにより，最大8個しかない格納できないレジスタスタックの値を，9個以上格納できるようにすることができます．

● 命令ポインタとデータポインタ

FPUは，FPUから例外割り込みが発生したとき，例外処理ハンドラが，例外を発生させた演算命令やデータの情報が取得できるようにするための命令をもっています．そのための情報を格納しているのが，この命令ポインタとデータポインタです．命令ポインタあるいはデータポインタの内容を見ることで，例外を発生させたFPU命令やそのアドレス（命令ポインタ），アクセスしたデータのアドレス（データポインタ）が取得できます．

命令ポインタやデータポインタは単体ではアクセスすることはできません．いままで述べてきたコントロールレジスタ，ステータスレジスタ，タグワードと一緒にアクセスすることになります．この一連の情報は，メモリにストアされますが，その形式はCPUの動作モード（リアル，プロテクト，16ビット，32ビット）により異なります．

FPU命令の種類と特徴

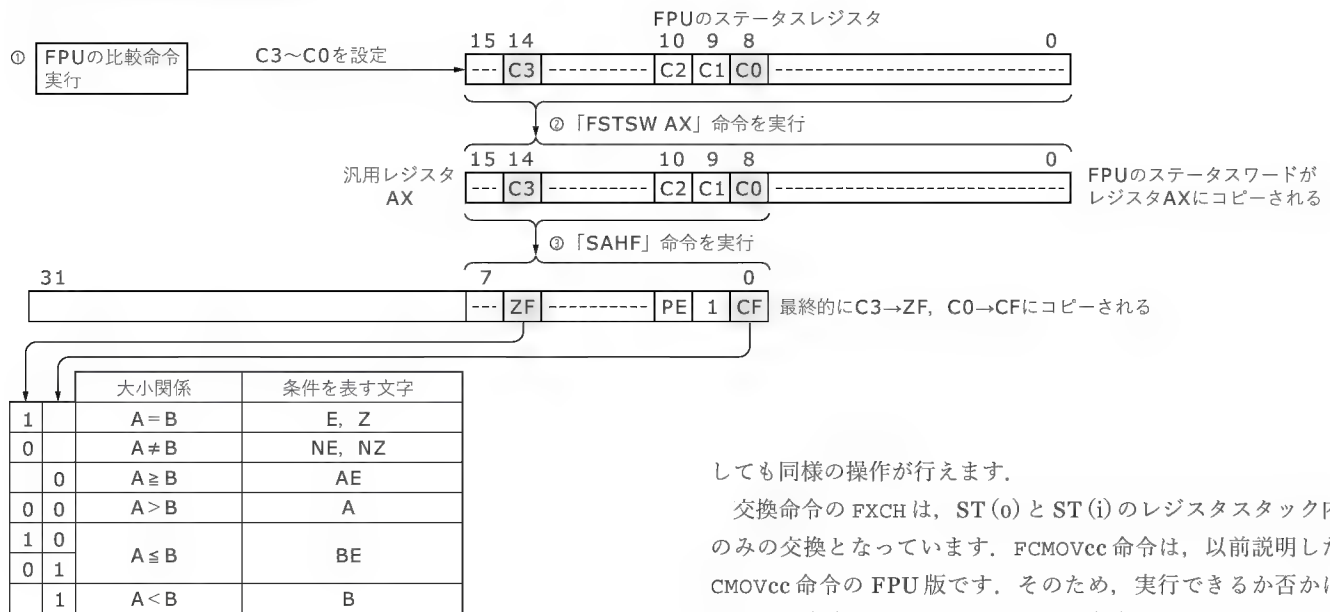
FPU命令には「データ転送」，「基本演算命令」，「比較命令」，「超越関数」，「定数ロード」，「x87 FPU制御命令」の6種類があります．**表2**は，この分類にしたがい，すべてのFPU命令を示したものです．

FPU命令で使われるニモニックには，文字の使われ方に一定の規則があります．それは，FPU命令はWAIT命令を除くすべての命令はFで始まるということです．また，データ転送，基本演算命令，比較命令では，Fの次にIがあればメモリ上にある整数を扱うことを意味し，Fの次がBならメモリ上にあるパックドBCDを扱うことを意味します．

ニモニックの最後がPで終わる命令は，命令実行終了後，レジスタスタックはPOPされ，それまでST(1)であった値が新たなST(0)になります．ニモニックの最後がPPの2文字で終わる命令は，命令実行終了後，レジスタスタックは2回POPされ，それまでST(2)であった値が新たなST(0)になります．

FPU命令のオペランドは，レジスタスタック間の転送演算では，転送元（SOU）と転送先（DEST）の二つのオペランドをもちます．ただし，この場合，転送元あるいは転送先のどちらかは

〔図6〕**FPUの比較結果の取得方法**

ZF	CF	大小関係	条件を表す文字
1		A＝B	E，Z
0		A≠B	NE，NZ
	0	A≧B	AE
0	0	A＞B	A
1	0	A≦B	BE
0	1		
	1	A＜B	B

ST(0)にする必要があります．たとえば，「ST，ST(4)」や「ST(1)，ST」の指定はOKですが，「ST(2)，ST(4)」のような指定はできません．

アセンブラMASMでは「DEST，SOU」と記述し，gasでは逆に「SOU，DEST」と記述します．また，gasでFPU命令を使う場合の注意ですが，このレジスタスタック間の転送演算では，ニモニックに型を示す文字は付きません．

アクセス対象がメモリや汎用レジスタの場合は，オペランドは一つとなります．その場合も，転送や演算，比較の命令では，もう一方のアクセス対象はレジスタスタックST(0)となります．

アセンブラMASMでメモリ上の値は，シンボルに型が入っているため，そのままシンボルをオペランドに記述します．gasでは，シンボルに型が入っていないので，オペランドだけでは指定されたメモリ上の値の型がわかりません．そこで“s”，“l”，“t”，“q”の文字のいずれかをニモニックの最後に付けます．単精度なら“s”，倍精度なら“l”，拡張精度“t”となります．そして，符号付き2進整数のワードは“s”，ショートは“l”，ロングは“q”です．パックドBCDは使用できる命令が決められているので型を示す文字は付きません．転送や演算，比較の命令でオペランドのない命令は，レジスタスタックST(0)，あるいはST(0)とST(1)が操作対象となります．

● データ転送命令

ロード，ストア，交換などを行います．ニモニックに“LD”があるのがロード，“ST”があるのがストアです．ロードは指定メモリ上の値を拡張精度に変換してレジスタスタックにPUSHし新たなST(0)とします．ストアではST(0)の値を指定された形式の値に変換しメモリにストアします．ロード，ストアではメモリ上の値の代わりに，レジスタスタック上のST(i)に対しても同様の操作が行えます．

交換命令のFXCHは，ST(0)とST(i)のレジスタスタック内のみの交換となっています．FCMOVcc命令は，以前説明したCMOVcc命令のFPU版です．そのため，実行できるか否かはCMOVcc命令と同じようにCPUID命令で調べます．また，MASMでFCMOVcc命令を使用する場合は，CPUの指定を「.686」とします．データ転送命令の実際のMASMでの記述例を**リスト1**，gasでの実際の記述例を**リスト2**に示しました．

● 基本演算命令

四則演算と剰余，平方根，絶対値，符号の付け替えといった演算を行います．ニモニックに“ADD”があるのが加算，同じように“SUB”が減算，“MUL”が乗算，“DIV”が除算となります．

減算と除算にはオペランドの演算順序があるため，オペランドの演算順序を逆にするための命令も用意されています．減算と除算のニモニックの後のほうに，“R”が指定されている命令がそれです．この減算と除算の命令は，MASMとgasでは動作が異なるので注意が必要です．MASMは，“R”なしが「DEST←DEST op SOU」，“R”付きが「DEST←SOU op DEST」となります．しかし，gasでは“R”なしが「ST(0) op ST(i)→DEST」，“R”付きが「ST(i) op ST(0)→DEST」となります．その例として減算および除算におけるMASMとgasの違いを**リスト1**，**リスト2**に示しました．

● 比較命令

レジスタスタック上の値を調べたり，ST(0)と他のレジスタスタック上の値との比較，ST(0)とメモリ上の値とを比較します．

比較命令には，クワイエット型NaNの扱いの違いにより二つの種類があります．連載第3回(2001年2月号)の実数の説明で，非数(NaN)があったと思います．非数はその名のように数ではありません．そして，非数には「クワイエット型NaN(QNaN)」，「シグナル型NaN(SNaN)」がありました．FPUは，SNaNに対する演算を無効オペレーションの例外とします．しかし，QNaNに対する演算は例外ではなく，結果として値を求める命令では不定実数(内容はQNaN)を生成します．

ニモニックに“U”の文字が付かない比較命令は，オペランド

にNaN(QNaNとSNaN)が指定されていた場合は無効オペレーション例外を発生します．しかし，ニモニックに“U”の文字が付いた比較命令は，オペランドのQNaNは無効オペレーション例外の対象とはしなくなります．

最初にも述べたように，現在CPUとFPUは見かけ上一つのチップに入っていますが，ハードウェア的にはCPUとFPUは独立しています．そのため，比較命令の結果(大小関係)は，FPUのステータスレジスタのC0～C3のフラグに設定されます．

しかし，この状態のままでは汎用命令の条件ジャンプなどの命令で使用することができません．そこで，FPU制御命令のFSTSW命令を使い，FPUのステータスレジスタの内容を汎用レジスタのAXに転送し，さらにSAHF命令でCPUのフラグレジスタに転送することで初めて，条件ジャンプなどに使用することができます(**図6**)．

● 超越関数

ST(0)の値に対して三角関数や指数関数，対数関数の演算を行います．FPUの超越関数は，**表2**に示すとおり8種類しかありません．そのため，これ以外の関数はこの8種類の命令から計算することになります．**表3**は，この8種類のFPU命令から一般的によく使われている三角関数，指数関数，対数関数，双曲線関数の計算式を示したものです．

● 定数ロード

ゼロや1，πといった定数をレジスタスタックにPUSHし，ST(0)に設定します．

● x87 FPU制御命令

この命令は，浮動小数点演算ユニット(FPU)を制御するための命令です．このFPUの制御は，OSやコンパイラ言語のライブラリといったルーチンが行うため，OSの下で動作しコンパイラ言語とリンクされるアセンブラのプログラムでは，ほとんどが勝手に使用することはできません．そのため，WindowsやLinuxで上で実行されるプログラム，それもコンパイラ言語とリンクされるアセンブラのプログラムでは，このFPU制御命令は使用しません．

もし間違って実行するとプログラムの実行に支障をきたしたり，誤った演算結果となる場合もあるので注意してください．ただし，例外としてFWAIT命令は，一般アプリケーションでも使用する必要がある重要な命令だといえます．

(1) FWAIT命令と例外

FPU命令のFWAIT命令は，本来CPUにあるWAIT命令の別名です．FPU命令と一緒に使用されることからFWAIT命令としてFPU命令に分類されています．

FWAIT命令は，FPU内で保留となっている未処理の例外を調べ，もし未処理の例外があればそれを処理するための命令です．FPUが命令を処理中に例外を発見した場合は，ステータスレジスタの例外フラグを設定します．例外がマスクされていなければ割り込み発生となるのですが，実際にCPUがFPUからの割り込みを検知するのは，例外を発生したFPU命令実行後，最初に実

〔表3〕FPUによる関数の計算式

● sinはFSIN命令，cosはFCOS命令，tanはFPTAN命令で求められる．
● $\tan^{-1}$はFPATAN命令で求める．$\tan^{-1}(\frac{y}{x})$=FPATAN(y, x)
● それ以外は次の式で求める．

$$\sin^{-1}(x)=\tan^{-1}\frac{x}{\sqrt{(1-x)(1+x)}}=\mathrm{FPATAN}(x,\sqrt{(1-x)(1+x)})$$

$$\cos^{-1}(x)=2\times\tan^{-1}\frac{\sqrt{1-x}}{\sqrt{1+x}}=2\times\mathrm{FPATAN}(\sqrt{1-x},\sqrt{1+x})$$

(a) 三角関数

● 指数関数はF2XM1命令で計算，x^yはFYL2X命令を使用する．

$2^x=(2^x-1)+1=\mathrm{F2XM1}(x)+1$
$e^x=(2^{x\times\log_2 e}-1)+1=\mathrm{F2XM1}(x\times\mathrm{FLDL2E})+1$
$10^x=(2^{x\times\log_2 10}-1)+1=\mathrm{F2XM1}(x\times\mathrm{FLDL2T})+1$
$Y^x=(2^{x\times\log_2 y}-1)+1=\mathrm{F2XM1}(\mathrm{FYL2X}(x, y))+1$

(b) 指数関数

● 対数関数はFYL2X命令で計算

$\log_2 x=1\times\log_2 x=\mathrm{FYL2X}(\mathrm{FLD1}, x)$
$\log_e x=\log_e 2\times\log_2 x=\mathrm{FYL2X}(\mathrm{FLDLN2}, x)$
$\log_{10} x=\log_{10} 2\times\log_2 x=\mathrm{FYL2X}(\mathrm{FLDLG2}, x)$

(c) 対数関数

$$\sinh(x)=\frac{\mathrm{sign}(x)}{2}\left[(e^{|x|}-1)+\frac{e^{|x|}-1}{e^{|x|}}\right]$$

$e^{|x|}-1=\mathrm{F2XM1}(|x|\times\mathrm{FLDL2E})=z$ とすると

$$\sinh(x)=\frac{\mathrm{sign}(x)}{2}(z+\frac{z}{z+1})$$

$$\cosh(x)=\frac{1}{2}(e^{|x|}+\frac{1}{e^{|x|}})=\frac{1}{2}\left[(z+1)+\frac{1}{z+1}\right]$$

$$\tanh(x)=\mathrm{sign}(x)\times(\frac{e^{2|x|}-1}{e^{2|x|}+1})$$

$e^{2|x|}-1=\mathrm{F2XM1}(2\times|x|\times\mathrm{FLDL2E})=y$ とすると

$$\tanh(x)=\mathrm{sign}(x)\times\frac{y}{y+2}$$

(d) 双曲線関数

$$\sinh^{-1}(x)=\mathrm{sign}(x)\times\log_e(z+1)$$
$$=\mathrm{sign}(x)\times\mathrm{FYL2XP1}(\mathrm{FLDLN2}, z)$$
$$z=|x|+\frac{|x|}{\frac{1}{|x|}+\sqrt{1+(\frac{1}{|x|})^2}}$$

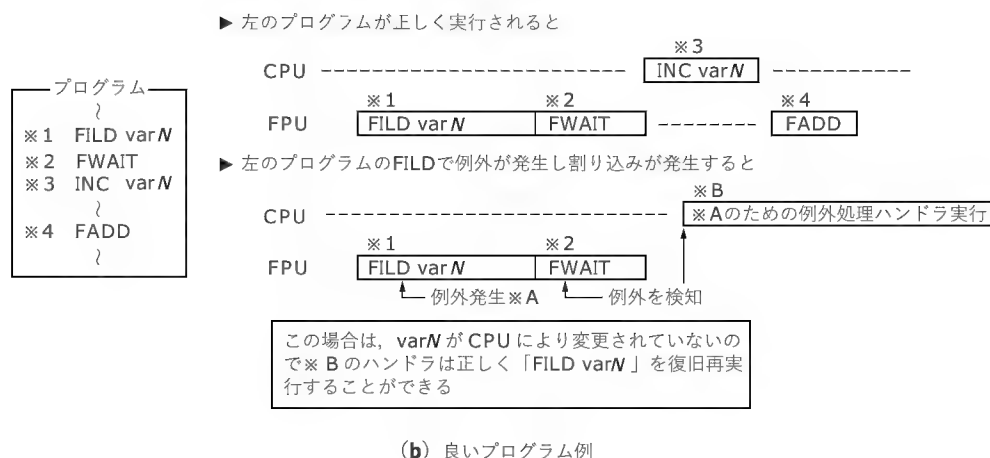
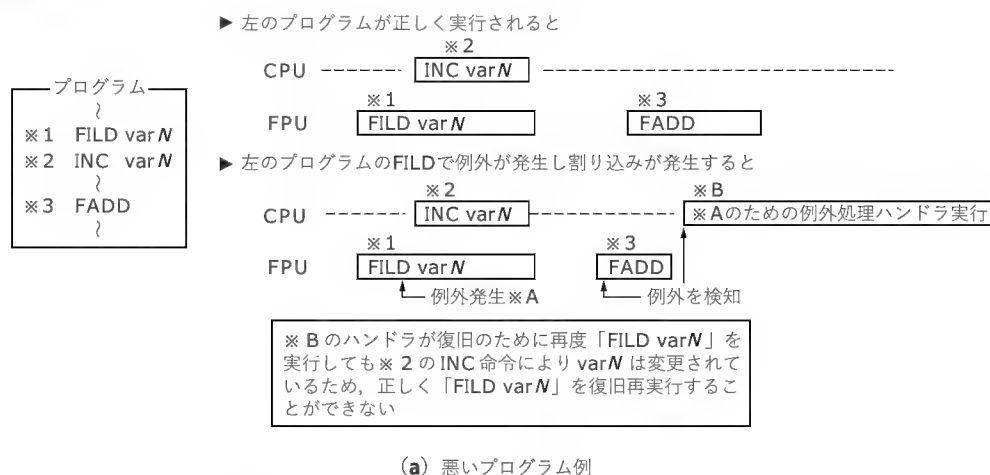
$$\cosh^{-1}(x)=\log_e(z+1)=\mathrm{FYL2XP1}(\mathrm{FLDLN2}, z)$$
$$z=x-1+\sqrt{(x-1)(x+1)}$$

$$\tanh^{-1}(x)=\frac{\mathrm{sign}(x)}{2}\times\log_e(z+1)$$
$$=\frac{\mathrm{sign}(x)}{2}\times\mathrm{FYL2XP1}(\mathrm{FLDLN2}, z)$$
$$z=\frac{2|x|}{1-|x|}$$

(e) 逆双曲線関数

〔図7〕FWAIT命令の使用例

（a）悪いプログラム例

（b）良いプログラム例

行されるFWAIT命令かFPU命令となります．そのため，例外発生と割り込み発生の間には時間差が生ずることがあります．

x87 FPU制御命令の一部を除き，FPU命令は本来の動作（転送や演算など）の前にFWAIT命令と同一の動作を自動的にします．そのた，FPU命令の前にFWAIT命令を付ける必要はありません．ただし，x87 FPU制御命令の内，FNで始まる命令は，意図的に本来の動作を始める前のFWAIT動作をしません．そのため，FWAIT動作をさせた場合に限り，x87 FPU制御命令のFNで始まる命令を使用することができます．

（2）FWAIT命令によるCPUとFPUの同期

前述したようにCPUとFPUは並行して動作しています．ただし，CPUとFPUが同じメモリ上の値をアクセスしてもプロセッサが自動的に同期を取るため，メモリ上の値はCPUもFPUも正しくアクセスできます．しかし，どうしても人為的にCPUとFPUで同期を取る必要があります．

具体的には，**図7**のようにFPU命令がリードしたメモリ上の値を，すぐにCPUが変更するプログラムで，メモリリードするFPU命令で例外が発生し，割り込みハンドラが起動したとき，不具合が生ずる場合があります．

通常，1命令の実行時間はCPUよりFPUのほうがかかります．そのため，FPUが1命令を実行している間に，CPUは数命令実行することになります．そのため，FPU命令で例外が発生した場合，割り込みが発生し，FPUの例外処理ハンドラの起動するまでの間に，CPUが命令を数命令実行し，FPUがリードしたメモリ値を変更している場合があります．これでは，せっかくFPUの例外処理ハンドラで復旧処理をしても，肝心の復旧に必要なFPU命令がリードするメモリ上のデータがCPUにより変更されていたのでは，正しく，例外を復旧することができません．

これを防ぐため，FPUがリードしたメモリ上の値を，すぐにCPUが変更する場合，FPUのメモリアクセス命令とCPUのメモリアクセス命令の間には，必ずFWAIT命令を入れ，FPUとCPUの同期を取るようにする必要があります．

＊　＊

次回は，x86系CPUがもつMMX命令について説明する予定です．

おおぬき・ひろゆき　大貫ソフトウェア設計事務所

初級ドライバ開発者のための

新連載

Windowsデバイスドライバ開発テクニック

第1回　デバイスドライバの基本関数

丸山治雄

本連載について

Windows NTが発売されて以来，Windows NT用デバイスドライバの解説書が，洋書，和書を問わず数多く出版されてきました．

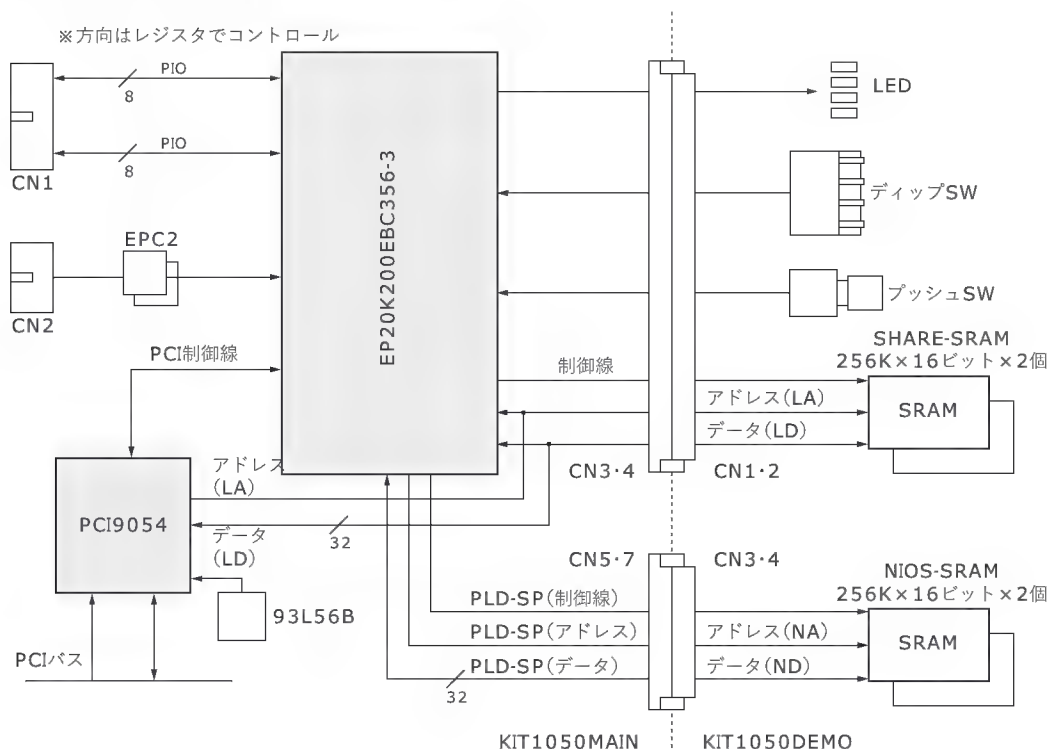
しかしながら，多くの書籍はDDK（Driver Development Kit）が提供している関数（DDK関数）の解説に終始し，関数の組み合わせ，すなわちプログラミング上の実践的な技巧に関する解説は皆無に近く，結局のところ試行錯誤しながらプログラミングすることになります．

本連載では，デバイスドライバ開発の初級者を対象に，具体的なデバイスドライバの開発を行いながら，DDK関数の使用方法を解説します．

なお，作成するドライバは，市販のPCIボード（KIT1050 PLX Getter II．**図1.1**，**写真1.1**）をターゲットとし，Windows NTとWindows 2000を共有するタイプとします．開発環境はWindows 2000 DDKとVisual C++ 6.0を使用し，すべてC言語で記述します．デバイスドライバのスケルトンを生成して細部を組み込むドライバの作成方法については，連載の最後の回で説明したいと思います．誌面の都合上，DDK関数の詳細はDDKのドキュメントを参照してください．

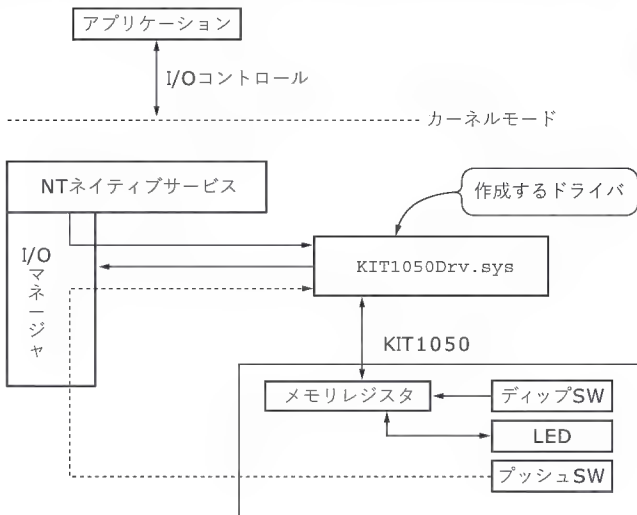
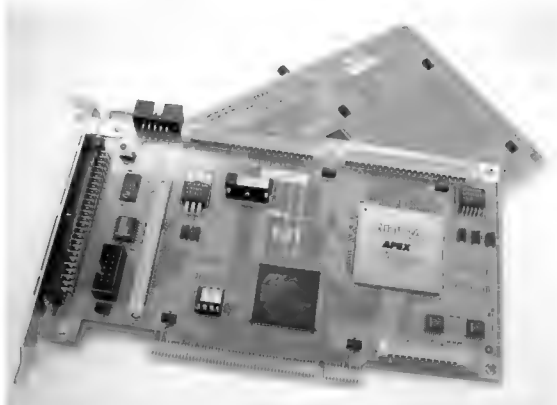
1.1　デバイスドライバに最低限必要な機能

今回説明するデバイスドライバのソースファイルの一覧を**表1.1**に示します．斜体の関数は，次回以降で説明する予定

〔図1.1〕KIT1050 PLX Getter IIのブロック図

〔表1.1〕デバイスドライバのソースファイル一覧

ファイル	内容	関数	関数の説明
Driverdef.h	デバイスドライバ用定義ファイル．ドライバで使用する，構造体などが定義されている		
Apidef.h kit1050def.h	アプリケーションとデバイスドライバ共通定義ファイル．KIT1050ボードのレジスタ定義，PLX9054のレジスタ定義，アプリケーションとドライバ間の通信用構造体が定義されている		
DriverEntry.c	ドライバの開始と停止を制御する本体部分で次の関数が記述されている	DriverEntry()	ドライバの登録処理を行う
		KIT1050 CreateClose()	アプリケーションがドライバ（デバイス）をオープン/クローズしたときに呼び出される関数で，メモリマップの作成，KIT1050ボードのレジスタ設定などを行う
		KIT1050Unload()	ドライバを停止するときに呼び出される関数で，割り込み，デバイスオブジェクトなどの削除や，ドライバ内部のメモリを削除する
		KIT1050Dispatch()	アプリケーションから要求された各種処理を行う
		KIT1050Cleanup()	アプリケーションから登録されたリリース要求待ちのイベントを削除する
PCIFIND.C		FindKIT1050_PCI()	ドライバが処理するPCIボードを探す
		GetPCIConfig()	PCIコンフィグレーションレジスタの内容を読み出す
		SetPCIConfig()	PCIコンフィグレーションレジスタにデータを書き込む
MEMORYMP.C		*MapMemMap TheMemory()*	PCIボードのメモリをアプリケーションでアクセスできるようにマッピングを行う
PCIRESOU.C	Windows NT専用．この関数内の機能を使用することはないと思われるので，説明は省略	PCIReportResource Usage()	PCIボードのリソースをシステムに登録
		PCIRemoveResource()	PCIボードのリソースをシステムから削除

※斜体の関数は次回以降で説明する

〔図1.2〕Windowsシステム，およびアプリケーションとデバイスドライバの関係

〔写真1.1〕KIT1050 PLX Getter II（ケーアイテクノロジー）の外観

です．また，Windowsシステム，およびアプリケーションとデバイスドライバの関係を**図1.2**に示します．

デバイスドライバを作成するときに最低限必要な機能（関数）を以下に示します．

● **DriverEntry()**

デバイスドライバのエントリ関数で，デバイスドライバ登録時に，この関数に制御が渡ってきます．

この関数内では，PCIボードを探して，必要リソースを読み込む処理，レジストリからドライバ起動に必要なパラメータを読み込む処理，`Create`関数，`Close`関数および`Unload`関数をシステムに登録するといった処理を行います．

● **Unload関数**

デバイスドライバを削除するときに必要になります．`DriverEntry()`の中で指定します．

● **Create関数，Close関数**

アプリケーションから`CreateFile()`によりドライバをオープンまたは，`CloseHandle()`によりクローズするときに，必要になります．

1.2 DriverEntry()の詳細

● レジストリ（パラメータ）のパス作成（**リスト1.1**）

`DriverEntry()`に制御が渡ってきたら，レジストリからドライバの動作に必要なパラメータを取得します．

レジストリのパスは，①の`RegistryPath`にセットされています．たとえば“`¥registry¥machine¥system¥ControlSet001¥services¥KIT1050DRIVER`”のようなパス名がUnicodeで格納されています．ここで，`ControlSet001`は固定の名前ではなく，そのときの状態により変わります．

②で`RegistryPath`にUnicodeで16個の文字数を加算しているのは，③で定義している`Parameters`を付加する分を確保するためです．最終的なレジストリパス名は，④で作成しています．

〔リスト 1.1〕レジストリのパス作成
(DriverEntry.c/DriverEntry()から)

```
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING RegistryPath ←①
    )
{

    RTL_QUERY_REGISTRY_TABLE paramTable[5]; ←⑥
    PDEVICE_OBJECT deviceObject = NULL;
    PDEVICE_EXTENSION  pExtension ;
    NTSTATUS       ntStatus;
    WCHAR   deviceNameBuffer[] = L"¥¥Device¥¥KIT1050DRIVER";
    UNICODE_STRING deviceNameUnicodeString;
    UNICODE_STRING deviceLinkUnicodeString;
    UNICODE_STRING    InRegistryPath;
    UNICODE_STRING    Parameter;
    WCHAR             ParameterKey[] = L"¥¥Parameters"; ←③
    UNICODE_STRING    ParameterPathUnicodeString;
    ULONG    InitLED;
    ULONG    DefaultData;
    PVOID    lockPtr;    // Lock Ptr Unload routine


    lockPtr = MmLockPagableCodeSection( KIT1050Unload );←⑤

// レジストリ(パラメータ)のパス作成
    InRegistryPath.MaximumLength = RegistryPath->MaximumLength +
                                   sizeof(WCHAR) * 16; ←②
    InRegistryPath.Length = 0;
    if (InRegistryPath.Buffer = ExAllocatePool(
                    NonPagedPool,
                    InRegistryPath.MaximumLength
                   ))
    {
        RtlCopyUnicodeString(&InRegistryPath, RegistryPath );
        RtlInitUnicodeString( &Parameter, ParameterKey );      ←④
        RtlAppendUnicodeStringToString(&InRegistryPath,
                                          &Parameter);
    }
    else
    {
        ntStatus = STATUS_INSUFFICIENT_RESOURCES;
        goto DriverExit;
    }
```

⑤でUnload関数(KIT1050Unload)をロックしているのは，もしDriverEntry()内でエラーが発生し(通常はPCIボードが装着されていないことによるエラー)，ドライバの登録を中止するときに，Unload関数がページアウトされないようにするためです．したがって，DriverEntry()を終了するときには，アンロックする必要があります．

もし，RegistryPathの名称を使用せずに，プログラム内で固定して使用するときは，**リスト1.2**のようにします．

● レジストリ(パラメータ)の取得(**リスト1.3**)

次にレジストリからパラメータを取得します．読み込む値は以下の四つです．

● **InitLED**

ドライバが正常に登録されたときに，KIT1050のLEDを点灯させる値(KIR1050のLEDは，電源投入直後はすべて点灯する．内部のレジスタ値は0x00)

● **Win2000**

Windows 2000であるかそれともWindows NTであるかを識別する値．Windows NTではリソース情報をドライバが登録する必要があるのでOSの識別が必要

〔リスト 1.2〕RegistryPathを使用せずにプログラム内で固定して使用する場合のルーチン

```
WCHAR ParameterPath[] =
  L"¥¥registry¥¥machine¥¥system¥¥CurrentControlSet¥¥
      services¥¥KIT1050DRIVER¥¥Parameters";

UNICODE_STRING ParameterPathUnicodeString;

// レジストリ(パラメータ)のパス作成
RtlInitUnicodeString(&ParameterPathUnicodeString,
                                             ParameterPath);

InRegistryPath.MaximumLength =
  ParameterPathUnicodeString.MaximumLength + sizeof(WCHAR);

InRegistryPath.Length = 0 ;
if (InRegistryPath.Buffer = ExAllocatePool(
           NonPagedPool,
           InRegistryPath.MaximumLength
          ))
{
  RtlCopyUnicodeString(&InRegistryPath,
                             &ParameterPathUnicodeString );
}
else
{ //メモリ不足
  ntStatus = STATUS_INSUFFICIENT_RESOURCES;
  goto DriverExit;
}
```

〔リスト 1.3〕レジストリの取得
(DriverEntry.c/DriverEntry()から)

```
// レジストリ(パラメータ)の取得
RtlZeroMemory(
            &paramTable[0],
            sizeof(paramTable)  ←①
            );

DefaultData = 0x0f;
paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;// Init. LED
paramTable[0].Name = L"InitLED";
paramTable[0].EntryContext = &InitLED;
paramTable[0].DefaultType = REG_DWORD;
paramTable[0].DefaultData = (PVOID)&DefaultData;
paramTable[0].DefaultLength = sizeof(ULONG);

DefaultData = 0x01;
paramTable[1].Flags = RTL_QUERY_REGISTRY_DIRECT;// Interrupt
paramTable[1].Name = L"Interrupt";
paramTable[1].EntryContext = &EnableInt;
paramTable[1].DefaultType = REG_DWORD;
paramTable[1].DefaultData = (PVOID)&DefaultData;
paramTable[1].DefaultLength = sizeof(ULONG);

DefaultData = 0x00;
paramTable[2].Flags = RTL_QUERY_REGISTRY_DIRECT;
                                      // Interrupt Shared
paramTable[2].Name = L"IntExclusive";
paramTable[2].EntryContext = &IntExclusive;
paramTable[2].DefaultType = REG_DWORD;
paramTable[2].DefaultData = (PVOID)&DefaultData;
paramTable[2].DefaultLength = sizeof(ULONG);

DefaultData = 0x00;
paramTable[3].Flags = RTL_QUERY_REGISTRY_DIRECT;// Win2000
paramTable[3].Name = L"Win2000";
paramTable[3].EntryContext = &Win2000;
paramTable[3].DefaultType = REG_DWORD;
paramTable[3].DefaultData = (PVOID)&DefaultData;
paramTable[3].DefaultLength = sizeof(ULONG);

RtlQueryRegistryValues(
   RTL_REGISTRY_ABSOLUTE | RTL_REGISTRY_OPTIONAL,
   InRegistryPath.Buffer,
   &paramTable[0],                                ←②
   NULL,
   NULL
   );

// レジストリ・パス名領域解放
ExFreePool( InRegistryPath.Buffer );
```

〔リスト 1.4〕**KIT1050NT.INI**

```
¥Registry¥Machine¥System¥CurrentControlSet¥Services¥KIT1050DRIVER
    Type = REG_DWORD 0x00000001
    Start = REG_DWORD 0x00000003   ←――――①
    Group = Extended Base
    ErrorControl = REG_DWORD 0x00000001
    Parameters
    InitLED = REG_DWORD 0x0f
    EnableInt = REG_DWORD 0x01
    IntExclusive = REG_DWORD 0x01
    Win2000 = REG_DWORD 0x00
```

〔リスト 1.5〕**KIT1050Drv.inf**（抜粋）

```
;; KIT1050Drv.inf

;--------- Version Section ---------------------------------------------------

[Version]
Signature = $Windows NT$
Provider=%ProviderName%

; If device fits one of the standard classes, use the name and GUID here,
; otherwise create your own device class and GUID as this example shows.

Class = Unknown        ;Other Devices
ClassGUID = {4d36e97e-e325-11ce-bfc1-08002be10318}
DriverVer=02/10/2003

;--------- ClassInstall/ClassInstall32 Section -------------------------------

; Not necessary if using a standard class

; 9X Style
[ClassInstall]
Addreg=Class_AddReg

; NT Style
[ClassInstall32]
Addreg=Class_AddReg

[Class_AddReg]
HKR,,,,%DeviceClassName%
HKR,,Icon,,"-18"

;---------- DDInstall Sections -----------------------------------------------
; --------- Windows NT -----------------

[KIT1050Drv_DDI.NT]
CopyFiles=KIT1050Drv_Files_Driver
AddReg=KIT1050Drv_NT_AddReg

[KIT1050Drv_DDI.NT.Services]
Addservice = KIT1050Drv, 0x00000002, KIT1050Drv_AddService

[KIT1050Drv_AddService]
DisplayName    = %SvcDesc%
ServiceType    = 1                  ; SERVICE_KERNEL_DRIVER
StartType      = 3                  ; SERVICE_DEMAND_START
ErrorControl   = 1                  ; SERVICE_ERROR_NORMAL
ServiceBinary  = %10%¥System32¥Drivers¥KIT1050Drv.sys

[KIT1050Drv_NT_AddReg]
HKLM, "System¥CurrentControlSet¥Services¥KIT1050DRIVER¥Parameters",¥
    "initLED", 0x00010001, 0x0f
HKLM, "System¥CurrentControlSet¥Services¥KIT1050DRIVER¥Parameters",¥
    "EnableInt", 0x00010001, 0x01
HKLM, "System¥CurrentControlSet¥Services¥KIT1050DRIVER¥Parameters",¥
    "IntExclusive", 0x00010001, 0x01
HKLM, "System¥CurrentControlSet¥Services¥KIT1050DRIVER¥Parameters",¥
    "Win2000", 0x00010001, 0x01
```

- **EnableInt**

Windows NTでは割り込みリソースを共有できないことが多いので，Windows NTでテストするときのために割り込みを使用するか否かの値

- **IntExclusive**

割り込みを共有するか否かの値

①のRtlZeroMemory()は，パラメータを読み込む領域の初期化を行っています．このとき，paramTable[5]（**リスト 1.1**の⑥）としてあったのは，読み込むパラメータの数＋1が必要だからです．②のRtlQueryRegistryValues()は，すべて0の領域で読み込みを終了します．

レジストリパス名領域は，OSがWindows 2000のときリソー

〔リスト 1.6〕デバイスオブジェクトの作成
(DriverEntry.c/DriverEntry()から)

```
// Device objectの作成
RtlInitUnicodeString (&deviceNameUnicodeString,
                      deviceNameBuffer);

ntStatus = IoCreateDevice (DriverObject,          ┐
           sizeof(DEVICE_EXTENSION),              │
           &deviceNameUnicodeString,              │
           FILE_DEVICE_MAPMEM,                    │
           0,                                     ├── ①
           FALSE,                                 │
           &deviceObject                          │
           );                                     ┘

if ( !NT_SUCCESS(ntStatus) )
    return( ntStatus );

//
deviceObject->Flags |= DO_DIRECT_IO ;   ←── ②
pExtension = deviceObject->DeviceExtension ;
RtlZeroMemory(
            pExtension,
            sizeof( DEVICE_EXTENSION )
            );

pExtension->DeviceObject = deviceObject ;
pExtension->InterfaceType = PCIBus;
pExtension->BusNumber = 0;
pExtension->InterruptObject = NULL;
pExtension->nCount = -1 ;
pExtension->TimerBusy = FALSE;
pExtension->TotalTimeout = FALSE;

// APIからCreateFile()によりOpenをかけるドライバー名
// WCHAR    deviceLinkBuffer[] = L"¥¥DosDevices
                                            ¥¥KIT1050DRIVER";
RtlInitUnicodeString(&deviceLinkUnicodeString,
                     deviceLinkBuffer);

ntStatus = IoCreateSymbolicLink (&deviceLinkUnicodeString,  ┐
                                &deviceNameUnicodeString);  ┘── ③

if (!NT_SUCCESS(ntStatus))
    goto DriverErrorExit;

// KIT1050 ボードの検出
ntStatus = FindKIT1050_PCI( deviceObject, &Information );  ←── ④
// レジストリーパス名領域開放
ExFreePool( InRegistryPath.Buffer );  ←── ⑤
// Clase Name領域の開放
ExFreePool( Information.DeviceClassName.Buffer );  ←── ⑥

if (!NT_SUCCESS(ntStatus))
    goto DriverErrorExit;
```

WDMを使わない理由

Windows用デバイスドライバには，複数の形態があります．それは，Windows NT専用ドライバ，Windows NT，Windows 2000共有ドライバ，そしてWDM(Windows Driver Model)で作成するWindows 2000，Windows 98/Me共用ドライバです．

Windows NT専用ドライバとWindows NT，Windows 2000共有ドライバを作成するときは，基本的にはWindows NT用ドライバを作成して使用します．WDMで作成するWindows 2000，Windows 98/Me共用ドライバは，一見すると開発が簡単なようですが，いくつかの制約があるので注意が必要となります．

したがって，筆者は，Windows 2000専用のデバイスドライバであっても，着脱可能なデバイス(PCMCIA CardBusなど)を除いてできるだけWDM手法は採用しません．

Windows 2000専用ドライバについては，Windows NT用ドライバにPnP機能を付加する方法を推奨しています．

〔リスト 1.7〕アプリケーションプログラム側の記述

```
HANDLE   hPCIHdl = NULL;     // ドライバハンドル
hPCIHdl = CreateFile("¥¥¥¥.¥¥KIT1050DRIVER",
    GENERIC_READ | GENERIC_WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_OVERLAPPED,         // 割り込み通知を受け取るときは必須
//    FILE_ATTRIBUTE_NORMAL,      // 割り込み通知は受け取れない
    NULL );

if ( hPCIHdl == INVALID_HANDLE_VALUE )
{
    hPCIHdl = (HANDLE)NULL;
    return( NULL );
}
```

スの取得のために使用するので，PCIボードのリソースを取得してから解放します．

ここで，レジストリにパラメータを登録する方法を説明します．

Windows NTでは，`KIT1050NT.INI`(**リスト 1.4**)を`RegIni.Exe`を使用してレジストリに登録します．デバイスドライバは，`%SystemRoot%¥SYSTEM32¥DRIVERS`に直接コピーします．その後システムを再起動し，デバイスマネージャでKIT1050DRIVERを"開始"にすると，ドライバの登録が開始されます．

①の`Start = REG_DWORD 0x00000003`は，マニュアルでドライバを開始させることを意味しています．これを`0x00000002`とすると，Windows NTが起動したときに自動的に開始されます．

もし，ドライバに問題があるとブルースクリーンになり，システムが起動できなくなる可能性があるので，ドライバ開発時には，必ず`0x00000003`としてテストすることを推奨します．

Windows 2000のときは，`KIT1050Drv.inf`(**リスト 1.5**)をデバイスドライバの登録ウィザードの中で指定します．

● デバイスオブジェクトの作成(**リスト 1.6**)

デバイスオブジェクトは，原則としてPCIボード1枚に一つのオブジェクトを作成します．今回は1枚のボードを対象にしているため特別な処理は行っていませんが，必要なら後で説明する**リスト 1.12**(p.145)のように複数枚のボードを処理できるように改造してください．

①の`IoCreateDevice()`は，システムがデバイスを識別するためにデバイス名，デバイスタイプを指定します．デバイス名は，アプリケーションから`CreateFile()`でドライバをオープンするときの名称ではありません．デバイスタイプは今回，

〔リスト1.8〕PCIボードの検出(PCFIND.C/FindKIT1050_PCI()から)

```
// KIT1050_PCIボードの有無を確認
for( BusNm = 0 ; BusNm < MAX_BUSES_TO_SEARCH ; BusNm++ )
{
  for( SlotNm = 0 ; SlotNm < PCI_MAX_DEVICES ; SlotNm++ )
  {
    RtlZeroMemory( &RegInfo,
                        (ULONG)sizeof( PCI_COMMON_CONFIG ) );
    PCISlot.u.AsULONG = 0;
    PCISlot.u.bits.DeviceNumber = SlotNm;
    status = HalGetBusData( PCIConfiguration,      ─┐
              BusNm,                                 │
              (ULONG)PCISlot.u.AsULONG,              ├←─ ①
              (PVOID)&RegInfo,                       │
              (ULONG)sizeof( PCI_COMMON_CONFIG )     │
              );                                    ─┘
    if ( status <= 4 )
                // 2=Invalid Vendor ID  4=No device this slot
      continue;
    // KIT1050_PCIボード情報を取得
    if ( ((USHORT)RegInfo.VendorID == (USHORT)PCIVendorID) &&
       ((USHORT)RegInfo.DeviceID == (USHORT)PCIDeviceID) )
    {
      pExtension->DeviceNode.BusNumber = BusNm; ←─ ②
      pExtension->DeviceNode.DeviceNumber = SlotNm;
      pExtension->DeviceNode.FunctionNumber = 0;
      pExtension->DeviceNode.Reserved = 0 ;
      pExtension->BusNumber = BusNm;

      // KIT1050_PCIボード情報を格納
      RtlCopyMemory( (PVOID)&pExtension->ConfigReg,
            (PVOID)&RegInfo,
            (ULONG)sizeof( PCI_COMMON_CONFIG )
            );

      PCIinf->pPCIMemAddr =
        pExtension->ConfigReg.u.type0.BaseAddresses[2] &
        0xfffffff0L;
      PCIinf->pPLXAddr =
        pExtension->ConfigReg.u.type0.BaseAddresses[0] &
        0xfffffff0L;
      PCIinf->pPLXioAddr =
        pExtension->ConfigReg.u.type0.BaseAddresses[1] &
        0xfffffffcL;
```

〔リスト1.9〕メモリ/ポートのサイズを取得(PCFIND.C/FindKIT1050_PCI()から)

```
if ( Win2000 == 0 )
{  // Windows NTのとき、メモリ/ポートのサイズを取得
  for ( BaseCnt = 0 ; BaseCnt < PCI_TYPE0_ADDRESSES ; BaseCnt++ )
  {
    BaseAddr = pExtension->ConfigReg.u.type0.BaseAddresses[BaseCnt]; ←─ ①

    // Get Mem/Port Size
    if ( (BaseAddr & 1) == 0 )
    {
      if ( (BaseAddr & 0xfffffff0) == 0 ) continue;
    }
    else
    {
      if ( (BaseAddr & 0xfffffffc) == 0 ) continue;
    }
    Limit = -1;
    SetPCIConfig( &pExtension->DeviceNode,
        (ULONG)(BaseCnt * 4 + 0x10), &Limit, 4 );   ←─ ②
    GetPCIConfig( &pExtension->DeviceNode,
        (ULONG)(BaseCnt * 4 + 0x10), &Limit, 4 );   ←─ ③
    SetPCIConfig( &pExtension->DeviceNode,
        (ULONG)(BaseCnt * 4 + 0x10), &BaseAddr, 4 ); ←─ ④
    if ( (BaseAddr & 1) == 0 )      ─┐
      Limit &= 0xfffffff0;            │
    else                              │
    {                                 ├←─ ⑤
      Limit |= 0xffff0000;            │
      Limit &= 0xfffffffc;            │
    }                                ─┘
    if ( BaseCnt == 0 )                          ─┐
      PCIinf->PLXMemSize = ~(Limit - 1);           │
    else                                           │
    if ( BaseCnt == 1 )                            ├←─ ⑥
      PCIinf->PLXioSize = ~(Limit - 1);            │
    else                                           │
    if ( BaseCnt == 2 )                            │
      PCIinf->MemSize = ~(Limit - 1);             ─┘
  }
}
else
{  // Windows 2000のときは、リソースをシステムから取得
  PCISlot.u.AsULONG = 0;                          ─┐
  PCISlot.u.bits.DeviceNumber = SlotNm;            ├←─ ⑦
  PCISlot.u.bits.FunctionNumber = 0;              ─┘
  status = HalAssignSlotResources(
    &InRegistryPath,
  //&Information->DeviceClassName,
    NULL,  // ClassNameはNULLでも構わない  ←─ ⑧
    Information->pDriverObject,
    Information->pDeviceObject,
    PCIBus,
    BusNm,
    PCISlot.u.AsULONG,
    &pAllocatedResources   ←─ ⑨
    );
  for (BaseCnt = 0 ;
       BaseCnt < (LONG)pAllocatedResources->List
                     ->PartialResourceList.Count;
       BaseCnt++ )
  {
    pResDescriptor = &pAllocatedResources->List                      ─┐
          ->PartialResourceList.PartialDescriptors[BaseCnt];          ─┘←─ ⑩
    // Check for port ranges
    if (pResDescriptor->Type == CmResourceTypePort)
    {  // Port
      PCIinf->PLXioSize = pResDescriptor->u.Port.Length;
    }
    else
    if (pResDescriptor->Type == CmResourceTypeMemory)
    {  // Memory
      if ( BaseCnt == 0 )  // PLX memory
        PCIinf->PLXMemSize = pResDescriptor->u.Memory.Length;
      else
      if ( BaseCnt == 2 )  // KIT1050 memory
        PCIinf->MemSize = pResDescriptor->u.Memory.Length;
    }
    else
    if (pResDescriptor->Type == CmResourceTypeInterrupt)
    {  // Interrupt
      pExtension->InterruptRes.Level =
        pResDescriptor->u.Interrupt.Level;
      pExtension->InterruptRes.Vector =
        pResDescriptor->u.Interrupt.Vector;
      pExtension->InterruptRes.Affinity =
        pResDescriptor->u.Interrupt.Affinity;
      pExtension->InterruptRes.Flags =
        pResDescriptor->Flags;
      pExtension->InterruptRes.Share =
        (CM_SHARE_DISPOSITION)pResDescriptor->ShareDisposition;
    }
  }
  ExFreePool( pAllocatedResources );  ←─ ⑪
}
```

FILE_DEVICE_MAPMEM(0x00008000)というDDKでは定義されていない値を使用しています．これは，PCIボードのメモリ領域をアプリケーションから直接アクセスできるようにするための宣言です(筆者は，UNIXの同等の機能からmmap，メモリマップと勝手に名づけている)．

②でDO_DIRECT_IOを指定していますが，この値を指定することにより，アプリケーションからReadFile()，またはWriteFile()の要求を処理できるようになります．ReadFile/WriteFileを使用しない場合は，DO_BUFFERED_IOを指定しても問題ありません．

③のIoCreateSymbolicLink()により，アプリケーションからオープンできる名称を作成します．今回の例では，アプリケーションから**リスト1.7**(p.141)のようにオープンします．この中でFILE_FLAG_OVERLAPPEDとFILE_ATTRIBUTE_NORMALの使い分けは，次回以降で説明する予定です．IoCreateSymbolicLink()で作成した名称はドライバをアンインストールするときに削除する必要があります．

デバイスオブジェクトを作成したら，次にPCIボードを探します．④でPCIボードを検出するFindKIT1050_PCI()を呼び出しています．⑤，⑥では**リスト1.1**，**リスト1.3**，**リスト1.6**で確保したメモリ領域を解放しています．

● PCIボードの検出(**リスト1.8**)

ここで，PCIボードを検出する関数FindKIT1050_PCI()について解説します．

①のHalGetBusData()でPCIスロット上にあるすべてのPCIデバイス情報を読み取ります．パラメータのPCIConfigurationは，DDKで定義されています．

バス番号とスロット番号を昇順に変えながらPCIスロットの情報を探します．関数の返り値が4以下のときは指定のバス，スロットに有効な情報がないことを示します．

返り値が5より大きいときは有効な情報を示すので，ベンダIDとデバイスIDが目的のボードのベンダIDとデバイスIDであるかを検査します．

KIT1050ボードはベンダID = 0x13d6，デバイスID = 0x0022なので，この両者の値が一致したものをドライバが管理するボードとします．

なお，今回のボードでは検査していませんが，マイクロソフト社は，サブベンダIDとサブシステムIDも検査の対象にするよう推奨しています．PCISIGでは，チップベンダとボードベンダを区別するために，仕様で明確に定義をしています．マイクロソフト社がWindows 2000になってから両方を検査するようにアナウンスをした目的は不明ですが，将来サブベンダIDとサブシステムIDがPCIボード識別のキー項目として使用される可能性があるので，できれば今から準備しておくのが安心かもしれません．

〔表1.2〕ベースアドレスのマッピング

ベースアドレス0	PLX9054のローカルコンフィグレーションレジスタのメモリアドレス
ベースアドレス1	PLX9054のローカルコンフィグレーションレジスタのポートアドレス
ベースアドレス2	PCIボードのメモリ/ポートアドレス1．KIT1050ボードの場合は，メモリレジスタにマッピングされている
ベースアドレス3	PCIボードのメモリ/ポートアドレス2．KIT1050ボードでは，未使用

⑦以降は，割り込みの登録，PCIボードのレジスタメモリ，またはメモリ空間をドライバあるいはアプリケーションで使用するために，目的のPCIボード情報を格納しておきます．

● メモリ/ポートのサイズを取得(**リスト1.9**)

以上で取得したPCIボードのリソースは，メモリまたはポートの先頭アドレスです．そこで，メモリサイズ，そしてWindows 2000の場合は物理割り込みベクタと論理割り込みベクタが必ずしも一致しているとは限らないため，論理割り込みベクタ(IRQ)を取得する必要があります．

なお，Windows NTの場合で，メモリサイズが事前にわかっているときは(通常はわかっているはずだが)**リスト1.9**の処理は省略してもかまいません．また，Windows 2000の場合で割り込みを使用しないときも省略してかまいません．しかし，割り込みを使用するときは，**リスト1.9**にあるように必ず論理IRQを取得する必要があります．

(1) Windows NTの場合

メモリまたは，ポートサイズを取得するときは，次のステップで取得します．詳細は，PCISIGの公式仕様書を参照してください[注1.1]．

1. ベースメモリアドレスの内容を保存します(①)．
2. ベースメモリアドレスに－1(0xFFFFFFFF)を書き込みます(②)．
3. ベースメモリアドレスの内容を読み取ります(③)．ここで読み取った内容がメモリ/ポートのサイズになります．
4. 「1」で保存した，ベースメモリの値を書き込みます(④)この処理は必ず行ってください．デバッガでステップ実行しているときに，この処理を行わないで処理を中止すると思わぬ結果になることがあります．
5. 「3」で得られたサイズ情報のうちメモリまたはポートの下位ビットをマスクします(⑤)．
6. サイズ情報は2の補数で表現されているので補数をとり，サイズとして必要な変数に格納します(⑥)．

KIT1050で使用されているPLX9054は，**表1.2**のようにベースアドレスがマッピングされています．

注1.1：PCISIGの仕様書などの情報は，PCISIGのメンバになれば簡単に入手できる．メンバになるだけなら，会費などは必要なく，オンラインサインアップのみで簡単にメンバになれる．

〔リスト 1.10〕物理アドレスから論理アドレスへの変換(PCFIND.C/FindKIT1050_PCI()から)

```
// 領域のマッピング
// メモリ空間
BusNm = pExtension->DeviceNode.BusNumber;

// Port
Size = PCIinf->PLXioSize;
Mem = PCIinf->pPLXioAddr;
PCIResources[2].PhysicalAddress.LowPart = Mem;
PCIResources[2].Length = Size;
mema.LowPart = (ULONG)Mem;
mema.HighPart = 0L;
Mem = (ULONG)0x0001;
HalTranslateBusAddress( PCIBus, BusNm, mema, &Mem, &memb );①
PCIinf->PLXioAddr = memb.LowPart;

// PLX Mem
Size = PCIinf->PLXMemSize;
Mem = PCIinf->pPLXAddr;
PCIResources[0].PhysicalAddress.LowPart = Mem;
PCIResources[0].Length = Size;
mema.LowPart = (ULONG)Mem;
mema.HighPart = 0L;
Mem = (ULONG)0x0000;
HalTranslateBusAddress( PCIBus, BusNm, mema, &Mem, &memb );①
PCIinf->PLXAddr = (ULONG)MmMapIoSpace(
memb,
Size,
FALSE
);

// PCI Mem
Size = PCIinf->MemSize;
Mem = PCIinf->pPCIMemAddr;
PCIResources[1].PhysicalAddress.LowPart = Mem;
PCIResources[1].Length = Size;
mema.LowPart = (ULONG)Mem;
mema.HighPart = 0L;
Mem = (ULONG)0x0000;
HalTranslateBusAddress( PCIBus, BusNm, mema, &Mem, &memb ); ←①
PCIinf->PCIMemAddr = (ULONG)MmMapIoSpace(
memb,
Size,
FALSE
);

// ドライバが使用する論理アドレス
pExtension->PCIRegPointer =
(PKIT1050_REGISTER)(PCIinf->PCIMemAddr + KIT1050REGISTER_
                                                  OFFSET);      ┐
pExtension->PCI9054RegPointer = (PPCI_PLX_CONFIG)PCIinf->       ├←②
                                                  PLXAddr;      ┘

// KIT1050 Board Reset
pExtension->PCIRegPointer->CTRL_0 = CTRL0_BRST;
// KIT1050 Int. Reset
pExtension->PCIRegPointer->CTRL_1 = 0;

// PLX9054のマスタ割り込み禁止
// Shared Run Time Register (Local Interrupt Enable bit off)
pExtension->PCI9054RegPointer->SHARED_ICS &= ~PCI90X0_LOCALINTENABLE;

return( STATUS_SUCCESS );

}
```

〔リスト 1.11〕
リソースをシステムに登録
(DriverEntry.c/DriverEntry()から. Windows NTの場合)

```
  // ResourceはWin2000では登録の必要なし
  if ( Win2000 == 0 )
  {                                                      ┐
    if ( !PCIReportResourceUsage (DriverObject,          │
              &PCIResources[0],                          │
              PCI_NUMBER_RESOURCE_ENTRIES                │
                        ) )                              │
    {                                                    ├←①
      ntStatus = STATUS_INSUFFICIENT_RESOURCES;          │
      goto DriverErrorExit;                              │
    }                                                    │
  }                                                      ┘

  // LEDを初期値で点灯する
  InitLED &= LEDMASK;                                    ┐
  pExtension->PCIRegPointer->LED = InitLED;              ┘←②

  //
  // 処理に必要なディスパッチ関数を登録
  //
  DriverObject->MajorFunction[IRP_MJ_CREATE]         = KIT1050CreateClose;  ┐
  DriverObject->MajorFunction[IRP_MJ_CLOSE]          = KIT1050CreateClose;  │
  DriverObject->MajorFunction[IRP_MJ_READ]           = KIT1050Read;         │
  DriverObject->MajorFunction[IRP_MJ_WRITE]          = KIT1050Write;        ├←③
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]= KIT1050Dispatch;      │
  DriverObject->MajorFunction[IRP_MJ_CLEANUP]        = KIT1050Cleanup;      │
  DriverObject->DriverUnload                         = KIT1050Unload;       ┘

DriverExit:
    MmUnlockPagableImageSection( lockPtr );              ┐
    return( ntStatus );                                  │
                                                         │
DriverErrorExit:                                         ├←④
  IoDeleteDevice( deviceObject );                        │
    MmUnlockPagableImageSection( lockPtr );              │
    return( ntStatus );                                  ┘
}
```

(2) Windows 2000の場合

Windows 2000の場合，リソースはシステムから取得します．PCIボードでは，メモリ/ボードのベースアドレスがボードのベースアドレスレジスタと変わることはありません．メモリ/ポートサイズもWindows NTの取得方法に比べ簡単です．

ただし，割り込みベクタ(IRQ)に関しては，IRQが他のボードと共有することがあるため(というよりほとんど共有になるので)，必ずシステムからリソースを取得し，システムが割り

〔リスト1.12〕複数枚のボードを制御するときのルーチン

```
// 複数枚のボードを処理するとき
for ( Board = 0 ; Board < 4 ; Board++ )
{
  // Device objectの作成
  RtlInitUnicodeString (&deviceNameUnicodeString,
                    deviceNameBuffer[Board]);
  ntStatus = IoCreateDevice (DriverObject,
             sizeof(DEVICE_EXTENSION),
             &deviceNameUnicodeString,
             FILE_DEVICE_MAPMEM,
             0,
             FALSE,
             &deviceObject
             );

  if ( !NT_SUCCESS(ntStatus) )
      return( ntStatus );

  Information.pDriverObject = DriverObject;
  Information.pDeviceObject = deviceObject;
  Information.DeviceClassName.MaximumLength = 0;
  Information.DeviceClassName.Length = 0;
  Information.DeviceClassName.Buffer = NULL;

  // KIT1050 ボードの検出
  // 同じボードを検出しないようにする必要があります
  ntStatus = FindKIT1050_PCI( deviceObject, &Information );
  // Clase Name 領域の開放
  ExFreePool( Information.DeviceClassName.Buffer );
  if (!NT_SUCCESS(ntStatus))
  {
    IoDeleteDevice( deviceObject );
    continue;
  }

  //
  deviceObject->Flags |= DO_DIRECT_IO ;
  pExtension = deviceObject->DeviceExtension ;
  RtlZeroMemory(
        pExtension,
        sizeof( DEVICE_EXTENSION )
        );
  pExtension->DeviceObject = deviceObject ;
  pExtension->InterfaceType = PCIBus;
  pExtension->BusNumber = 0;
  pExtension->InterruptObject = NULL;
  pExtension->nCount = 0;
  pExtension->TimerBusy = FALSE;
  pExtension->TotalTimeout = FALSE;

  // APIからCreateFile()によりOpenをかけるドライバ名
  // WCHAR  deviceLinkBuffer[] = L"¥¥DosDevices¥¥KIT1050DRIVER";
  RtlInitUnicodeString(&deviceLinkUnicodeString,
                       deviceLinkBuffer);

  ntStatus = IoCreateSymbolicLink (&deviceLinkUnicodeString,
                              &deviceNameUnicodeString[Board]);
  if (!NT_SUCCESS(ntStatus))
    goto DriverErrorExit;
}
```

当てたIRQを使用する必要があります．

⑦は，HalAssignSlotResources()に引き渡すスロット情報をPCISlot構造体に設定しています．ここで必要なのはスロット番号とファンクション番号ですが，複数のファンクション番号を設定したボードでなければ，ファンクション番号は0でかまいません．

HalAssignSlotResources()に引き渡すClassNameは，NULLでもとくに問題はありません(⑧)．リソース情報は，pAllocatedResourcesに格納されます(⑨)．各リソースタイプにより，サイズとIRQを取得します．DDKの構造体が複雑なので，詳細はDDKのヘッダファイルをじっくり見てください．⑩の変数pResDescriptorにPCIボードのリソース情報が入っています．

pAllocatedResourcesは，リソースの取得処理が終わったら必ず解放してください(⑪)．

● 物理アドレスから論理アドレスへの変換(**リスト1.10**)

最後に，取得したPCIボードのメモリおよびポートの物理アドレスをドライバ内でアクセスできる論理アドレスに変換します．

ここで変換した論理アドレスは，あくまでもドライバ内で使用するもので，アプリケーションで使用するメモリマッピングではないことに注意してください．

①のHalTranslateBusAddress()で，Mem変数にはベースメモリアドレスの内容が，ポートのときは1，メモリのときは0を与えて，変換結果はmembの64ビットアドレス構造体に格納されます．

そして，pExtension->PCIRegPointerとpExtension->PCI9054RegPointerがドライバ内で使用するKIT1050ボードのメモリアドレスとPLX9054のローカルコンフィグレーションレジスタのメモリアドレスになります(②)．

PCIボードの検出が終わったら，Windows NTの場合はリソースをシステムに登録します(**リスト1.11**の①)．

KIT1050ボードに実装されているLEDをレジストリパラメータの内容で点灯させます(②)．最後に，ドライバが処理する機能をシステムに登録します(③)．

DriverEntry()が呼び出されたときにロックしたUnload関数のロックを解除して，DriverEntry()の処理は終わりになります(④)．

複数枚のボードを制御するときはボード単位にデバイスオブジェクトを作成し，シンボリックリンク名をたとえば，KIT1050DRIVER0，KIT1050DRIVER1……のようにします．**リスト1.12**は簡単なサンプルです．アプリケーションから複数のドライバをオープンするときは**リスト1.13**のようにします．

〔リスト1.13〕アプリケーションプログラム側の記述

```
if ( Board == 0 )
  strcpy( string, "¥¥¥¥.¥¥KIT1050DRIVER0" );
if ( Board == 1 )
  strcpy( string, "¥¥¥¥.¥¥KIT1050DRIVER1" );

hPCIHdl = CreateFile( string, GENERIC_READ | GENERIC_WRITE,
```

＊　　　　＊

Unload関数，Create/Close関数については，次回解説する予定です．

まるやま・はるお　ドライバ屋

第22回

日本語でプログラミングを行う開発環境 — TTSneo

開発環境探訪

水野貴明

今回紹介する「TTSneo」は，日本語でプログラミングを行う開発環境である．TTSneoは，日本語でプログラミングができるというだけでなく，タートルグラフィックやTTSneo自体をマクロとして他のアプリケーションに組み込む機能など，さまざまな機能をもったユニークな開発環境となっている．

インストール

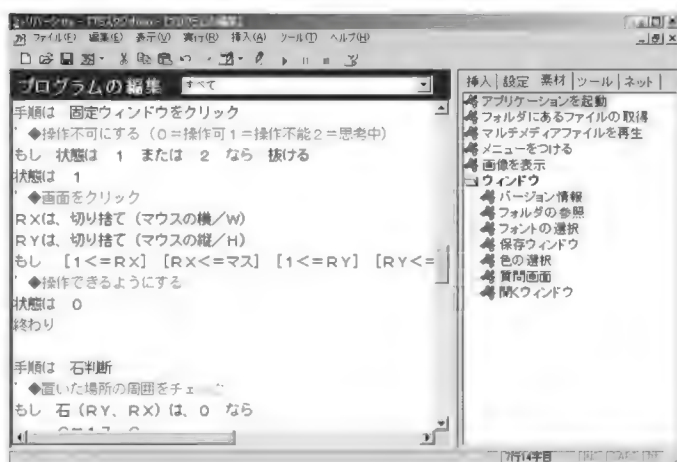
TTSneoは標準的なインストーラ形式になっており，インストール作業は非常に簡単である．ただし，TTSneoはVisual Basicで作られており，OSの標準ではないコンポーネントをいくつか必要とするため，そのままインストールしただけでは動かない場合がある．そういった場合は，TTSneoの配布元で配布されているVisual Basic 6.0ランタイムを別途インストールする必要がある．

TTSneoでの開発は，付属する「TTSスタジオneo」という統合開発環境(**図1**)上で行う．TTSスタジオneoは，プログラムの入力，実行，ブレークポイントの指定，ステップ実行など，基本的な機能を備えたかなり本格的な開発環境である．特徴的なのは「機能バー」と呼ばれる画面右側の領域で，命令を簡単に挿入するための「挿入」，よく利用する命令セットをまとめて挿入できる「素材」，カラーパレットから色を選択したり，TTSneoのアップデートなど，その他さまざまな機能を利用できる「ツール」など，開発をより便利に進めるための機能をたくさん盛り込んでいる．ツール自体もTTSneo自身で書かれており，自分で新しい機能を追加することも可能だ．また，「ネット」というタブには，IEコンポーネントを利用したブラウザ機能が組み込まれており，Webページを表示できる．開発中に必要な情報をWebから取得する，というのは非常によくあることなので，いちいちブラウザに切り替えずにWebページを参照できるこの機能は，プログラミング時の思考の分断を少なくでき，なかなか便利な機能だといえるだろう．

DATA

名称：TTSneo
作者：ゆうと氏
Webサイト：`http://hp.vector.co.jp/authors/VA021321/index.html`
現在のバージョン：0.72(安定版)，0.74(開発版)[2003/07/16現在]
ダウンロードサイズ：2.7Mバイト(インストーラ)

〔図1〕TTSneoの統合開発環境「TTSスタジオneo」

作成したプログラムは，TTSスタジオneo上でも実行できるほか，EXEファイルに変換することもできる．作成されたEXEファイルは，プログラム＋ラインタイムルーチンという形になっており，Visual Basic 6.0ランタイムさえインストールされていれば，他のマシンで実行することもできる．

TTSneoのプログラミング仕様と機能

簡単なTTSneoのプログラムを**リスト1**に示す．これは，指定したテキストファイルの句読点を「，」「．」から「、」「。」に置き換えて，新たに別のファイルとして保存するプログラムである．すでに述べているように，TTSneoのプログラムの最大の特徴は，日本語でプログラムを記述するという点であり，このサンプルプログラムもすべて日本語で書かれている．「データは」で

Column 作者に直撃インタビュー

TTSneoの作者ゆうと氏に，メールでのインタビューに答えていただいた．

Q：TTSneoを開発しようと思ったきっかけは何ですか？

A：パソコンの難しい知識がなくても，すぐに始められるプログラミング言語を作りたかったということが，いちばんの動機です．〝プログラムを作る〟というと，どうしても一般の人からは理解できない特別なことのように思われがちです．そうではなく，ワープロで文章を打つことや，グラフィックソフトで絵を描くことと同じように，プログラムを作るということがあってもいいのではないかと思っています．そのとき中学の授業で「ロゴライター」に出会って，日本語でソフトが作れたら便利ではないかと感じ，TTSを作り始めました．

Q：今後どのようなに発展させていく予定ですか？

A：うーん．まだ高速化や安定化といった大きな課題を抱えているので，まずはこれらの課題を解消していきたいと思います．それには，Visual Basic以外の言語で改めて書き直し，コンパイラの実装などもしていきたいと思います．今後ものんびりながら，改良を続けて成長させていきたいと思います．

Q：開発に際して苦労した点はどこですか？

A：いろいろあります．Visual Basicで作っているのでどうしても処理が遅く，どうしたら高速化できるか考えたり，ユーザーの要望を反映させるために，あの手この手でかなり無理をして機能を実装しています．また，開発を始めたのが中3の頃で，プログラミングに関する知識があいまいで，うまくコーディングできず，バグ修正にも苦労しました．

Q：作者として，TTSneoのここがすごい！という点を教えてください．

A：エディタです．TTSスタジオneoでは，VBライクなRAD環境が標準でサポートされ，コードを入れることなく，視覚的かつ短時間でウィンドウの作成ができます．そういった開発環境は，もちろんVisual Studio .NETやDelphiにもありますが，TTSはテンプレートなども備えているので，より簡単に使えるのではないかと思っています．

Q：作者として，TTSneoの弱点……という点を教えてください．

A：処理速度です．処理速度が速ければとても良い言語だという意見もたくさん寄せられています．今後，コンパイラを作ってこの弱点をなくしたいと思います．

Q：TTSneoは，どんなものを開発するのに向いていると思いますか？

A：エディタやツールなどの一般的なWindowsアプリケーションの開発です．私自身，実用的なアプリケーションを作るのが得意なので，命令もサンプルも実用的なツールを作るためのものが多くなっています．残念ながら処理が遅いので，大量データの処理や速度を要求するゲームを作るという用途にはあまり向いていないと思います．

Q：日本語でのプログラミングをすることの意義について，どう考えていますか？

A：やはり，英語に抵抗をもつことなくプログラミングできるという点は大きいと考えています．とはいっても，JavaやCといった言語と同じレベルでTTSがあるということではありません．TTSはプログラミングの「最初のステップ」であって，TTSでプログラミングの感覚を知ってもらい，プログラミングに興味をもったらJavaやC，.NETなどの，「次のステップ」へ進んでもらえればと思います．

といっても，TTSで十分だという方もおられますので，そういう場合はTTSでソフトを作るというのでOKです．そういうスタイルを私は考えています．

Q：フリーの開発環境の意義について，どう考えていますか？

A：気軽に始められ，いろいろな種類の言語を試すことができるということだと思います．昔は何万円もした高価なソフトを買わないといけなかったので，一つの言語に固まってしまい，いろいろな言語を使ってみるということが難しかったのですが，最近ではJavaも.NETも最低限の開発環境は，無料でダウンロードできるので，自分の使いやすい/用途に合ったプログラミング言語を選ぶことができるのが利点だと思います．

始まる3行以外は，比較的自然な日本語になっていることがわかる．

「データは」で始まる3行についてはやや意味がわかりづらいが，これらの行では「データ」という変数に値を代入する処理が行われている．「○○は　△△」という記述が，「○○」という変数に「△△」の示すデータを代入する，という処理を行うものである．そして，これらの行では代入元のデータが「開け（ファイルを開き内容を取得する関数）」「置き換え（文字列の置換を行う関数）」という関数の戻り値になっている．したがってこの3行は，ファイルからデータを「データ」という変数に読み込み，「，」を「、」に，「．」を「。」に置き換える処理を行っていることになる．

TTSneoにおける命令/関数は，「～を表示」「終了」といった体言止めの形と，「繰り返せ」「塗れ」といったような命令形のもの

〔リスト1〕「，」「．」を「、」「。」に全置換するプログラム

```
'TTSneoサンプルプログラム
'指定したファイルの「，」「．」を
'「、」「。」に全置換する

開くファイルを選択
もし　選択ウィンドウのファイル名　が　「」　なら　終了
データは　開け（選択ウィンドウのファイル名）
データは　置き換え（データ、「，」、「、」）
データは　置き換え（データ、「．」、「。」）
保存するファイルを選択
もし　選択ウィンドウのファイル名　が　「」　なら　終了
選択ウィンドウのファイル名　へ　データ　を保存
```

が存在する．なお，次のように命令の後に言葉を追加してあっても，問題なく解釈される．

「こんにちは」を表示しろ

TTSneoでは，プロシージャを作成することもできる．プロシージャは「手順は　○○（○○は名前）」〜「終わり」で囲むことで作成できる．次のように，再帰させることも可能である．

```
手順は　らせん
　　もし　長さ＞200　なら　ぬける
　　前へ　長さ
　　左へ　角度
　　長さは，長さ＋2
　　らせん
終わり
```

インタビューにもあるようにTTSneoの文法は，「自然に話すような日本語でプログラムを作る」ことを中心に考えられているわけではない．日本語はあくまで，プログラミングをわかりやすくするための手段としてとらえられており，TTSneoをマスタしたユーザーが，Visual BasicやJavaなどの異なる言語に移行する際に，できるだけスムーズに移行できることを第一に考えているとのことである[注1]．そのため，条件分岐などを利用する際も「もし　○は　□　なら」という書き方だけでなく，「もし　○＝□　なら」というBASICの「IF　○＝□　THEN」に相当する書き方ができるなど，互換性の高い書き方もサポートしている．また，書き方によってはあまり日本語としては正しくはなくなってしまうが，そもそも英語を利用しているBASICやC言語なども，正しい英語で記述できるわけではないので，そういったプログラミング言語における英語の役割を日本語に置き換えたものである，と考えるとわかりやすいかもしれない．

続いて，TTSneoに搭載されている機能だが，一般的な計算処理やファイル操作のほか，GUIの利用，動画や音楽の再生，タートルグラフィック，Direct3D，FTP/HTTPによるインターネットアクセス，正規表現など，幅広く用意されており，ホビーから実用的なものまで，さまざまなアプリケーション作成に対応している．また，プラグインやActiveXなどを利用して，新たに機能を追加することも可能になっており，C/C++やVisual Basicなどのほかの言語を知っていれば，それらの言語と連携させることも可能になっている．

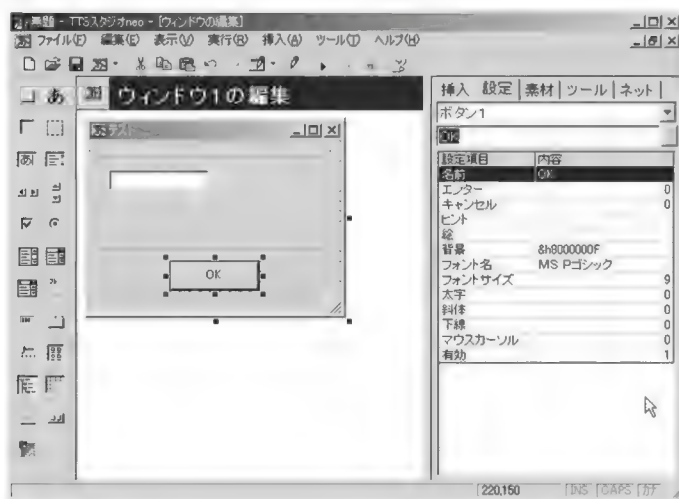
GUIプログラミングとオブジェクト

さまざまなTTSneoの機能の中でも特徴的なものとして，今回は「GUIプログラミング」，「タートルグラフィック」，「TTS For Application」を紹介する．まずは，GUIプログラミングである．

TTSでは，ウィンドウ上にボタンやテキストボックスを配置したGUIアプリケーションを作成することもできる．しかもTTSスタジオneoには，ウィンドウをデザインするツールがついており，非常に簡単に画面の設計を行える（**図2**）．このツールを利用してウィンドウをデザインすると，それが自動的にTTSneoのコードに変換されるようになっている（**リスト2**）．

TTSneoでは，GUIコントロールは「オブジェクト」として管理される．利用可能なオブジェクトは，テキストボックスやボタン，チェックボックスといったものから，IEブラウザコンポーネント，ツリービュー，リッチテキストなど，標準的なコントロールはすべて利用することができる．

オブジェクトを生成するコマンドは「作れ」である．オブジェクトごとに固有の名前を付けることはできず，**リスト2**のように，「オブジェクト　番号　を作れ」と指定することで，画面上にオ

〔図2〕TTSスタジオneoはウィンドウの編集も可能

〔リスト2〕TTSスタジオneoによって自動生成されたプログラム

```
手順は　ウィンドウ1の表示
ウィンドウ1を使う
'ウィンドウ1
ウィンドウ1の中の大きさを252，173へ変えろ
　　その名前を「テスト」へ変えろ
　　その背景を&h8000000Fへ変えろ
フレーム1を作れ
　　その位置を10，10へ変えろ
　　その大きさを230，100へ変えろ
　　その名前を「」へ変えろ
テキスト1を作れ
　　その位置を20，30へ変えろ
　　その大きさを100，20へ変えろ
ボタン1を作れ
　　その位置を80，120へ変えろ
　　その大きさを90，30へ変えろ
　　その名前を「OK」へ変えろ
ステータスバー1を作れ
　　その位置を0，153へ変えろ
　　その大きさを252，20へ変えろ
'ウィンドウ1終わり
ウィンドウ1を表示
終わり
```

注1：「TTSneoの開発指針」http://yutopia.s8.xrea.com/x/tech/point.html

〔リスト3〕タートルグラフィックで星を描くプログラム

```
タイトルは　「星を書く」
ウィンドウを表示
出てこい

ペンを上げろ
前へ　３０
左へ　９０
前へ　８０
右へ　１８０

ペンを下ろせ
５回繰り返せ
前へ　１６０
右へ　１４４
繰り返し終わり
```

〔図3〕タートルグラフィックで描いた星

ブジェクトが表示され，オブジェクトの種類と番号で，それぞれのオブジェクトを管理することになる．また，番号で管理するのが煩雑になる場合は，次のように各オブジェクトに別名をつけ，その名前で管理することも可能になっている．

　ボタン１をOKボタンとして作れ

GUIを利用したプログラミングのスタイルは，一般的なWindowsのプログラム言語と同様に，「ウィンドウ１のボタン１をクリック」，「ウィンドウ１のボタン１でマウスのボタンを上げる」といったイベントプロシージャをつくり，そこにそれぞれのイベントが発生した際の処理を記述する．また，オブジェクトの属性（プロパティ）を参照，変更する場合は，「『オブジェクト』［番号］の『プロパティ名』」という指定の仕方をする．たとえば次の例では，テキストボックスのサイズを変更するように指定している．

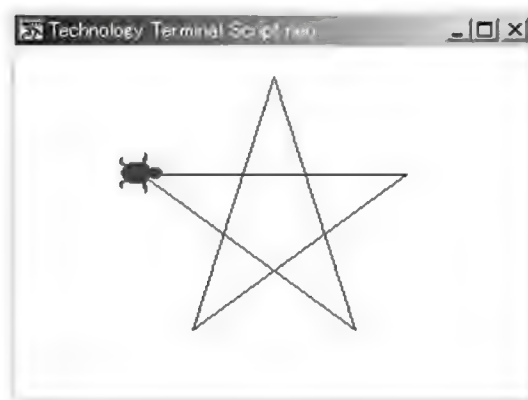
　テキスト１の大きさを100，20へ変えろ

イベントを受け付けるようにするには，「待機」という命令を利用する．「待機」はいわゆるイベントループを生成する命令だ．TTSneoのプログラムは，通常は先頭から1行ずつ実行されていくが，この命令が実行されると，プログラムの実行はそこで停止し，イベント待機モードに切り替わる．

なおTTSneoでは，GUIコントロールのほかに，印刷を行う「プリンタ」，MS Office製品を扱う「ワード」，「エクセル」などのオブジェクトも利用可能である．たとえば，次のサンプルはExcelを起動して，セルの値を変更するプログラムである．

　エクセルを起動
　エクセルのワークブックを追加
　エクセルのセル（A1）を，「あいうえお」に変えろ

こちらはGUIコントロールとは異なり，「作れ」で生成する必要はなく，また番号を指定する必要もない．

また，「拡張オブジェクト」という名前のオブジェクトは，ActiveXクラスやコントロールを呼び出して利用するためのオブジェクトである．これを利用すれば，そのマシンに登録されたActiveXはすべて利用可能である．

さらにTTSneoには，「その」，「この」という指示代名詞のオブジェクトが存在する．「この」はイベントプロシージャ内において，イベントが発生しているオブジェクトを指定するもので，こちらはJavaにおける「this」などと同様の働きをもつものである．一方「その」は，最後に操作したオブジェクトを示すもので，**リスト2**でも使われているが，同じオブジェクト名を何度も繰り返さずに，プログラムを記述することが可能になるものだ．

タートルグラフィック

TTSneoには「タートルグラフィック」という機能がある．これは，LOGOという言語に実装されていた機能で，タートル（カメ．線を描画するためのカーソルの役割をする）を使って線を描画していく機能だ．タートルには向きがあり，その移動は「前へ○歩進む」，「○度右に方向転換する」といった，現在のタートルの位置と向きを基準として移動の指定を行う．これを繰り返し命令と組み合わせて利用することで，さまざまな図形を描画できるわけだ．TTSneoは，ロゴライター[注2]という日本語を利用したLOGOの実行環境に影響を受けているとのことで，その流れからタートルグラフィックの機能が実装されている．

TTSneoでは，タートルはまさにカメの形をしており，「出てこい」という命令でウィンドウ上に表示することができる．タートルは「前へ」，「後ろへ」という命令で前後へ線を引き，「右へ」，「左へ」という命令で角度を指定することで，方向転換を行える．「ペンを上げろ」と命令すれば，移動しても線が描画されないようにすることも可能だ．ループ処理と組み合わせることで，複雑な模様も簡単に描ける．たとえば**リスト3**のように記述すれば，**図3**のような星を描くことができる．

サンプルプログラムに入っている「あみだくじ[注3]」では，ター

注2：ロゴジャパン（株）が販売するLOGOの実行環境．http://www.logo.co.jp/（現在アクセス不可）
注3：サンプルの投稿作品フォルダに入っている「あみだくじ.tts」

トルグラフィックを利用して，カメにあみだくじを解かせている(**図4**)．カメを単なるカーソルとしてだけでなく，このように画面上の登場キャラクタとして利用するのも面白い．ちなみに，カメを別のキャラクタに変更する機能もついている．

TTS For Application

TTSneoは，ランタイムルーチンのActiveX DLL版が用意されており，これを利用すると，ActiveXに対応した開発言語の中にTTSneoのランタイムを組み込んで，マクロのような働きをさせることができる．これは「TTS for Application(TTSA)」と呼ばれる．

TTSneoのActiveX DLL(`TTScript.dll`)は，本体とともにインストールされているが，TTSAに関するサンプルやドキュメントは，TTSneo本体とは別に，「TTSneo SDK」というセットとして配布されている．その中には，Visual Basic 6.0からTTSAを呼び出すためのサンプルなどが入っている．

Visual Basic 6.0からTTSAを呼び出すには，あらかじめ，`TTScript.dll`を参照するように設定しておき(プロジェクトメニューの「参照設定...」を選択し，「日本語プログラミング言語」にチェックを入れる)，**リスト4**のように記述する．指定するクラス名は「`TTScript.TTS`」だ．プログラムの実行には「`Run`」メソッドを利用する．ほかにもファイル名で指定したファイルを実行する`RunFile`メソッドなども用意されている．ただし，これらのメソッドは値を返すことができないので，TTSAから値を返したい場合には，実行後に`GetSetInfo`メソッドを利用して，変数の値を取り出す必要がある．

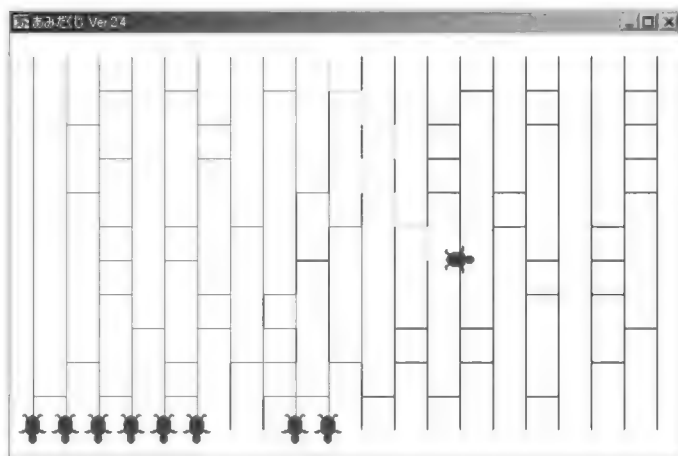
また，オブジェクト参照変数の定義の際に`WidthEvents`を指定し，「呼び出し元」というオブジェクト名を利用して，TTSAからコールバックを行うことができるようになる(**リスト5**)．Viaual Basic側では**表1**のような三つのイベントが用意されており，TTSAと呼び出し元が自由にデータのやり取りを行えるようになっている．**リスト5**の例では，TTSAのプログラムが実行されると，呼び出し元がコールバックされ，そのときに呼び出しに利用された名詞と動詞が(Visual Basic側で)表示される．

TTSAは，自作のアプリケーションにマクロとして組み込んだり，ExcelやWordのVBAから呼び出して利用するなどの利用方法が考えられる．TTSneoを採用することで，プログラミングにあまりなじみのない人にでも扱うことが比較的容易なマクロ機能を実装できるのではないだろうか．

なおTTSAを利用するには，Visual Basic 6.0のランタイムと`TTScript.dll`が必要になるため，TTSAを利用したアプリケーションを配布する場合には，このファイルを一緒に配布する必要がある．また，`TTScript.dll`ダウンローダという最新版のDLLをダウンロードしてくれるツールも公開されており，DLLファイル本体の代わりにこちらをつけることもできる．

おわりに

TTSneoの作者であるゆうと氏は，現役の大学生である．そして，TTSの開発を開始した当時は中学生だったそうだ．そういったこともあり，TTSneoは若い世代，もっといえば小中学生にでも，比較的容易にプログラミングが体験できるような配慮がたくさんされている．それは単に「日本語」でプログラミン

〔図4〕「あみだくじ」カメが問題を解いているところ

〔表1〕コールバック用に用意されたイベント

GetData	TTSAが，呼び出し元からデータを取得しようとした場合に発生する．例：××は、呼び出し元の○○
SetData	TTSAが，呼び出し元にデータを渡そうとしたときに発生する．例：呼び出し元の○○を「××」に変える
RumCmd	TTSAが，呼び出し元に処理を実行させたいときに発生する．例：呼び出し元の「○○」を××

〔リスト4〕Visual BasicからTTSAを呼び出しているところ

```
Private Sub Command1_Click()
    Dim cTTSA As TTScript.TTS
    Set cTTSA = New TTScript.TTS
    cTTSA.Run "「こんにちは」と表示", 0
End Sub
```

〔リスト5〕Visual BasicでTTSAからのコールバックを実現するプログラム

```
Option Explicit
Dim WithEvents cTTSA As TTScript.TTS

Private Sub Command1_Click()
    cTTSA.Run "呼び出し元の「何か」を探す", 0
End Sub

Private Sub cTTSA_RunCmd(Doush As String, Meish As String,
NotFound As Boolean)
    MsgBox Meish + "を" + Doush
End Sub

Private Sub Form_Load()
    Set cTTSA = New TTScript.TTS
End Sub
```

グが可能である，というだけではない．たとえばマニュアルでは，2匹のカメが登場し，その掛け合いでプログラミングの基礎が学べるしくみになっている(**図5**)．さらに，マニュアルは漢字に振り仮名が振られたものも公開されており，「難しい専門用語」や英語といった，プログラミングを学ぶ上での，プログラミング以外の障壁がなるべく低くなるように工夫されている．

そして事実，TTSneoのユーザーグループには，そういった若い世代の開発者が多く参加しており，前述のマニュアルの作成やプログラミングコンテストの開催など，活発な活動が行われ，楽しくプログラミングを行うことができる環境作りに，大いに貢献している．

筆者がプログラミングを学びだしたのも小学生の頃だったが，その頃はまだインターネットも普及しておらず，プログラミングに関する情報といえば大人向けのものばかりで，何かと苦労させられた覚えがある．もちろん，その苦労が楽しかった，という面もあるので，いちがいにそれを否定するわけではないが，TTSneoのような，若年層に向けてアピールされた言語が存在する状況は，うらやましくも感じる．

TTSneoは，インタビューで作者のゆうと氏も述べているように，速度の問題などの課題は残しているが，機能も豊富で，プログラミングを気軽に始められる環境としては，なかなかすぐれているといえるだろう．

日本語でのプログラミング環境を語ると，「はたして日本語でプログラミングを行う必要があるのか」という話題がついてまわるが，プログラミングを学ぶ上で重要なのは，自分(＝人間)のやりたいことを，プログラミング言語という「機械のわかる処理手順」に翻訳変換する作業であり，その作業のプロセスは，どんな言語であろうと関係ない．むしろ，「英語」という日本人にとって障壁となりやすい部分がないことで，そういったプログラミングの基礎をよりきちんと学べるようになるメリットもある．英語というだけでプログラミングを「何だか難しそう」としり込みしている人たちも，日本語なら，「これなら自分でもわかるかもしれない」という気になるかもしれない．そういう意味でも，日本語でプログラミングができるという選択肢があることは非常に重要で，意義のあるものではないだろうか，と筆者は考えている．

TTSneoでプログラミングを始めた人たちが，それをきっかけに，さまざまな分野で活躍してくれるようになることを，期待したい．

〔**図5**〕**TTSneoのチュートリアルマニュアル**

みずの・たかあき

Pentium4/Intel Xeonにおける性能モニタ機能を利用した

メモリプロファイリングツールを開発する ―― 実践編

吉岡弘隆

はじめに

この解説の前編(本誌2003年8月号)では，まず，Intel社の32ビットマイクロプロセッサ(IA-32と記す)のハードウェア性能モニタ機能について紹介し，次にIntel Pentium4/Intel Xeonプロセッサで強化された機能について詳細に解説した．後編となる今回は，Pentium4/Intel Xeonの性能モニタ機能を利用して実装したメモリプロファイリングツールについて，利用方法，実装などを解説する．具体的には，精密なイベントベースサンプリング(PEBS : Precise Event Based Sampling)により，アプリケーションの性能上のボトルネックが容易に発見できることを示す．

PEBSにより，割り込みハンドラを起動するレイテンシやそのオーバヘッドが削減され，しかもイベント発生の正確な特定と，その時点での精密なコンテキストを入手できるようになった．PEBSを利用することで，PentiumIIIなど従来のPentium系プロセッサでは不可能だった精密なサンプリングも可能になった．筆者らはPEBSの機能にいちはやく注目し，PEBSを利用したメモリプロファイリングツールを開発した．本稿では開発の動機や背景を紹介するとともに，メモリプロファイリングツールの実装およびそれを利用した性能測定と分析方法を解説する．

CPUの処理速度は年率数十%で向上しているものの，メモリの処理速度の向上は年率数%といわれている．つまり，メモリの処理速度はCPU処理速度の向上率に比べて低い．結果，CPUとメモリの速度のギャップは年々広がっている．たとえば最近のCPUでは，メモリアクセス(キャッシュミス)のペナルティは200倍近くあり，メモリアクセスのコストは一定という仮定のプログラミングモデルは破綻した．**表1**に，Intel Xeon 2GHz/メインメモリ1Gバイトでのアクセスコストの実測値を示す．より高性能なソフトウェアを実現するためには，メモリ階層を意識したプログラミングモデルが必要となってきた．

〔表1〕Intel Xeon 2GHz/メインメモリ1Gバイトでのアクセスコスト

メモリ階層	アクセスコスト
L1	1ns
L2	9ns
メインメモリ	196ns

たとえば，SPECベンチマークに代表される科学技術アプリケーションのCPI(Clock Per Instruction)に比べ，商用ワークロード(OLTP : On Line Transaction Processing)のCPIが大きいことが知られている．要因の一つは，メモリアクセス時のストールである．たとえばリレーショナルデータベースのメモリアクセスはポインタを使ったランダムアクセスが多いので，科学技術アプリケーションの単純な配列の順アクセスに対する最適化は，単純には適用できない．そこで，メモリアクセスに注目した性能向上ツールが必要とされている．

しかし，従来のタイマ割り込みによるプログラムカウンタ(PC : Program Counter)のサンプリング手法によるプロファイリングでは，キャッシュミスなど，メモリの動的特性に関する詳細情報は得られない．そのため開発者は，おおまかなホットスポット(実行時間を多く消費している場所)の位置を推定できても，なぜそこで実行時間がかかるかを特定するのが困難だった．たとえば，命令がストールしているのはデータメモリのキャッシュミスなのか？　分岐予測の失敗なのか？　それとも単に実行にコストがかかる命令を実行中だったのか？　……などに関する情報が得られない．さらに，最近のプロセッサではアウトオブオーダ実行，深いパイプラインや投機的実行などにより，イベントを発生させた命令とPCの値が必ずしも一致しないので，前述したプロファイリングだけで問題を特定することがますます困難になっている．

Pentium4/Intel Xeonは，この問題に対し，ハードウェアによる性能モニタ機能(18個の性能モニタカウンタ)を備え，さまざまなメモリアクセスイベント(L1/L2キャッシュミス/TLBミスなど)を測定できる．またPEBS機能により，以前のCPUでは不可能だった，より精密なプロセッサの状態のサンプリングも可能となった．そこで筆者らは，Pentium4/Intel XeonのPEBSを利用してメモリプロファイリングツールを実装し，リレーショナルデータベース(PostgreSQL)やLinuxカーネルのプロファイリングを行い，ツールの有効性を実証した．このツールを利用することで，メモリイベント(L1/L2キャッシュミスなど)のホットスポットを容易に発見できた[5),6)]．

1. メモリプロファイリングツール利用のための準備

筆者らは，Pentium4/Intel Xeonプロセッサのハードウェア性能モニタ機能を利用した，hardmeterというツールをGNU/Linuxプラットホーム上に実装した．ここでは簡単に，そのインストール方法および利用方法を紹介する．

1.1 インストール前の準備

hardmeterは，Pentium4/Intel Xeonプロセッサのみをサポートするので，インストールするマシンがそれを満たしているか確認する(**図1**)．**図1**のように，vendor_idがGenuineIntelでcpu familyが15であればよい．

1.2 インストールに必要なファイルの準備

hardmeterに必要なファイルは次のとおりで，①と②は次のWebページ(`http://sourceforge.jp/projects/hardmeter/`)にある．

① hardmeterのソースコード
② perfctrのソースコード
③ Linuxカーネルのソースコード

hardmeterが，今回開発したメモリプロファイリングツールの本体である．執筆時点での最新版は，hardmeter 2003-0603である．デフォルトのLinuxでは，IA-32のハードウェア性能モニタ機能が有効になっていない．そこで，それを有効にするカーネルパッチが必要で，筆者らはperfctrというパッチを使った．今回対応したperfctrは，2.5.4である．

Linuxカーネルのソースコードは，`http://www.kernel.org/`などから入手できる．

1.3 hardmeter対応カーネルの作成

● ソースの展開

ダウンロードしたソースを展開する．ここでは例として，Linux Kernel 2.4.20を利用する．

```
cd /usr/src
tar xvfj linux-2.4.20.tar.bz2
tar xvfz perfctr-2.5.4.tar.gz
tar xvfz hardmeter-030603.tar.gz
```

● パッチを当てる

perfctrにパッチを当てたうえで，perfctrのupdate-kernelを実行する．

(perfctrへのパッチ当て)

```
cd perfctr-2.5.4
patch -p1 < ../hardmeter-030603/patch/
                                perfctr-2.5.4.dif
```

(カーネルのperfctr対応)

```
cd ../linux-2.4.20
../perfctr-2.5.4/update-kernel
```

そのほか，作業するかどうかは任意だが，`linux-2.4.20/Makefile`の先頭の，

```
EXTRAVERSION =
```

を，

```
EXTRAVERSION = -hardmeter
```

と変更することをおすすめする．こうすると，`'uname -r'`を実行したときに，`'2.4.20-hardmeter'`と出力され，今動いているカーネルが`hardmeter`対応かどうかの確認がしやすくなる．

また，カーネルの分析のために，`CFLAGS`に`-g`(デバッグシンボル付きでコンパイルする)を追加しておくとよい．具体的には，

```
CFLAGS := $(CPPFLAGS) -Wall -Wstrict
            -prototypes -Wno-trigraphs -O2
          -fno-strict-aliasing -fno-common
```

を，

```
CFLAGS := $(CPPFLAGS) -Wall -Wstrict
            -prototypes -Wno-trigraphs -O2
       -fno-strict-aliasing -fno-common -g
```

にする．

● カーネルの作成とインストール

まずperfctrを有効にする．“make config”,“make menuconfig”,“make xconfig”のいずれかを実行して，

```
Processor type and features  →
Performance-monitoring counters support  →
```

とたどっていって，

```
Performance-monitoring counters support
Virtual performance counters support
```

を有効にする．ほかの項目は有効にしてもしなくてもよいが，hardmeterを使うには少なくとも`Virtual performance counters support`が有効になっている必要がある．

〔図1〕プロセッサの確認

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 1
model name      : Intel(R) Pentium(R) 4 CPU 2.00GHz
stepping        : 2
cpu MHz         : 1977.487
cache size      : 256 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8
apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
ss ht tm
bogomips        : 3945.26
```

〔図2〕ebsのコマンドシンタックスとサポートするイベントタイプ

```
Usage: ./ebs (-u | -k) [-o OUTFILE] [-i INTERVAL] [-c COUNT] -t TYPE EXE_OR_PID
  options
    -u           - sample user-mode events
    -k           - sample kernel-mode events
    -o OUTFILE   - store sampled data to file
    -i INTERVAL  - sampling interval(default: 10000)
    -c COUNT     - max sampling count(default: 2000)
    -t TYPE      - event type to sample
    -m NAME,NAME... - event masks
  help options
    -h           - show event types
    -h TYPE      - show event masks
```

(a) コマンドシンタックス

```
imprecise at-retirement event:
      instr_retired   - instruction retired
      uop_retired     - uops retired
precise front-end event:
      memory_loads    - memory loads
      memory_stores   - memory stores
      memory_moves    - memory loads and stores
precise execution event:
      packed_sp_retired - packed single-precision uop retired
      packed_dp_retired - packed double-precision uop retired
      scaler_sp_retired - scaler single-precision uop retired
      scaler_dp_retired - scaler double-precision uop retired
      64bit_mmx_retired - 64bit SIMD integer uop retired
      128bit_mmx_retired - 128bit SIMD integer uop retired
      x87_fp_retired  - floating point instruction retired
      x87_simd_memory_moves_retired - x87/SIMD store/moves/load uop retired
precise replay event:
      l1_cache_miss   - 1st level cache load miss
      l2_cache_miss   - 2nd level cache load miss
      dtlb_load_miss  - DTLB load miss
      dtlb_stor_miss  - DTLB store miss
      dtlb_all_miss   - DTLB load and store miss
      mob_load_replay_retired - MOB(memory order buffer) causes load replay
      split_load_retired - replayed events at the load port.
```

(b) サポートするイベントタイプ

また,

Processor type and features →

とたどっていって,「Symmetric multi-processing support」か「Local APIC support on uniprocessors」のどちらかを有効にする.上記二つの項目以外は,環境にあわせて任意に設定する.次に,カーネルとカーネルモジュールのコンパイルを行う.カーネルソースのルートディレクトリで,

```
make dep; make clean; make bzImage
make modules; make modules_install
```

を実行し,

```
cp arch/i386/boot/bzImage /boot/
                    vmlinuz-2.4.20-hardmeter
cp System.map /boot/
                    System.map-2.4.20-hardmeter
```

とする(linux-2.4.20/MakefileのEXTRAVERSIONを変えていない場合は-hardmeterは不要).

上記のコマンドで,カーネル本体とsystem mapを/bootの下にコピーする.次に,liloなどのブートローダの設定を変えて,

```
/boot/vmlinuz-2.4.20-hardmeter
```

のカーネルで再起動する.

カーネル再起動後,

```
# mknod /dev/perfctr c 10 182
# chmod 644 /dev/perfctr
```

とする.以上で,hardmeter対応カーネルの作成は終了である.

1.4 hardmeterユーザーアプリケーションのコンパイル

まず,'uname -r'を実行してhardmeter用に作成したカーネルが立ち上がっているかどうか確認する.次に,hardmeterのソースファイルのルートディレクトリに移動してmakeを実行する.src/ebs,src/libhardmeter.oが作成されたら,コンパイルは完了である.

2. メモリプロファイリングツールを使う

srcディレクトリにebs(イベントサンプリングツール)がある.ebsのコマンドシンタックスおよびサポートするイベントタイプを**図2**に示す.

〔図3〕' pwd 'を実行したときのユーザーモードでの1次キャッシュミスのイベントを1000回ごとにサンプリングする

```
$ ./ebs -u -i 1000 -t l1_cache_miss pwd
#eflags  liner_ip   eax      ebx      ecx      edx      esi      edi      ebp      esp
/usr/src/hardmeter-030603/src
00000246 40006edc bffff428 400124b8 00000000 bffff428 bffff428 4200bde8 bffff478 bffff3a0
00000246 40007a80 40000658 400124b8 400121e8 00000000 400121e8 00000000 bffff3e8 bffff3b0
00000283 40008827 00000000 400124b8 4212a3d8 00000007 4201513c 00000000 bffff4f8 bffff4b0
00000206 420261f2 00000000 4212a2d0 00000065 421185c0 4212e258 00000000 bffff5a8 bffff570
00000202 40009ff0 0d696910 400124b8 420080c8 40024020 000004d7 40012c30 bffff910 bffff8c8
#
# start time : Sun Jun 22 22:12:47 2003
# user       : yes
# kernel     : no
# interval   : 1000
# count      : 5
# tsc        : 1518880
#
# event name : l1_cache_miss
# description: 1st level cache load miss
# event mask : + nbogus
#              - bogus
#
```

〔図4〕生データ

```
$ head pebs
#eflags  liner_ip   eax      ebx      ecx      edx      esi      edi      ebp      esp
00000293 c0245012 f7bdcac8 f7bdca80 00000000 00000000 f7d02c00 0000001e 40000000 f679df04
00000206 c0142bba 00000b00 ef11ed00 00000b00 f63c8000 0000a9e1 ef11ed80 00000000 f63c9e74
00000202 c0142bba 00000b00 ed775100 00000b00 f63c8000 00009657 ed775180 00000000 f63c9e74
00000202 c0142bba 00000b00 ebef9200 00000b00 f63c8000 000082ce ebef9280 00000000 f63c9e74
00000202 c0142bba 00000b00 eac00780 00000b00 f63c8000 00006f52 eac00800 00000000 f63c9e74
00000206 c0142bba 00000b00 e8ca6780 00000b00 f63c8000 00005bc9 e8ca6800 00000000 f63c9e74
00000202 c0142bba 00000b00 e73b2d80 00000b00 f63c8000 0000483b e73b2e00 00000000 f63c9e74
00000206 c0142bba 00000b00 e6238980 00000b00 f63c8000 000034be e6238a00 00000000 f63c9e74
00000202 c0142bba 00000b00 e4108b80 00000b00 f63c8000 00002134 e4108c00 00000000 f63c9e74
```

2.1　メモリプロファイリングツールを利用した性能測定

図3の例では，pwdを実行したときのユーザーモード(-u)での1次キャッシュミス(-t l1_cache_miss)のイベントを1000回(-i 1000)ごとにサンプリングした．eflags，論理アドレス，各レジスタ値(eax/ebx/ecx/edx/esi/edi/ebp/esp)の値を取得できた．このとき，EXE_OR_PID〔コマンドないしはプロセスID(pid)〕を指定しなければ，システム全体のイベントの測定を行う(グローバルモード)．測定したいプロセスが複数にわたっている場合，非常に便利である．

2.2　プロファイリングデータの測定と分析

ここでは例として，Linux Kernel 2.4.20をビルドしたときのカーネルのキャッシュミスを測定してみよう．実験に利用したノートPCのスペックは，次のとおりである．

CPU ： Pentium4
メモリ： 1024Mバイト，DDR 266MHz

カーネルのビルドを行っている最中にhardmeterのebsで各種イベントを測定した．

```
$ ./ebs -k -t l1_cache_miss -c 50000
                                    -o pebs
```

これで，pebsというファイルにカーネルモードの1次キャッシュミス(l1_cache_miss)が出力される．

〔図5〕トップ10リスト

```
$ awk '{print $2}' pebs |sort|uniq -c|sort -nr|head
   3776 c0142bba
   2514 c0130d68
    750 c012d3a6
    324 c0245012
    297 c010915d
    295 c012e483
    268 c0138af4
    260 c01642a3
    251 c0244fcc
    191 c0141fa0
```

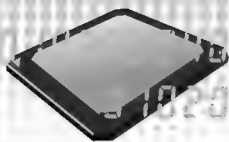
● **分析方法**

図4のような生データが取得されたとしよう．カーネルのどの部分でキャッシュミスが多発しているだろうか？

```
$ awk '{print $2}' pebs |sort|uniq -c|
                                sort -nr
```

2桁目が論理アドレスなので，それでソートして件数を数えると，トップ10リストを容易に作成できる(**図5**)．

論理アドレスとルーチン名のつきあわせは，eip2rというスクリプトをhardmeterに同梱しているので，それを利用する(**図6**)．

この例では，invalidate_bdevというルーチンのc0142bbaというアドレスで1次キャッシュミスが3776回発生しているということがわかる．このc0142bbaというアドレスでどのよう

〔図6〕論理アドレスとルーチン名のつきあわせ

```
$ awk '{print $2}' pebs |../doc/eip2r -s ../../linux-2.4.18-18.8.0-hm030603/System.map|sort|uniq -c|sort -nr|head
   3776 c0142bba invalidate_bdev
   2514 c0130d68 file_read_actor
    750 c012d3a6 __constant_memcpy
    324 c0245012 unix_poll
    297 c010915d ret_from_sys_call
    295 c012e483 find_vma
    268 c0138af4 rmqueue
    260 c01642a3 proc_pid_stat
    251 c0244fcc unix_poll
    191 c0141fa0 fget
```

〔図7〕当該アドレス付近の逆アセンブル結果

```
              for (i = nr_buffers_type[nlist]; i > 0 ; bh = bh_next, i--) {
c0142b95:     8b 34 ad 90 f7 39 c0    mov    0xc039f790(,%ebp,4),%esi
c0142b9c:     85 f6                   test   %esi,%esi
c0142b9e:     7e 27                   jle    c0142bc7 <invalidate_bdev+0x67>
c0142ba0:     c7 44 24 04 00 e0 ff    movl   $0xffffe000,0x4(%esp,1)
c0142ba7:     ff
c0142ba8:     ba 00 e0 ff ff          mov    $0xffffe000,%edx
c0142bad:     21 e2                   and    %esp,%edx
c0142baf:     89 54 24 04             mov    %edx,0x4(%esp,1)
                      bh_next = bh->b_next_free;

                      /* Another device? */
                      if (bh->b_dev != dev)
c0142bb3:     8b 44 24 08             mov    0x8(%esp,1),%eax
c0142bb7:     8b 7b 20                mov    0x20(%ebx),%edi
c0142bba:     66 39 43 0c             cmp    %ax,0xc(%ebx)
%%%%%%%% ここでキャッシュミスが多発している．
c0142bbe:     74 30                   je     c0142bf0 <invalidate_bdev+0x90>
c0142bc0:     4e                      dec    %esi
c0142bc1:     85 f6                   test   %esi,%esi
c0142bc3:     89 fb                   mov    %edi,%ebx
c0142bc5:     7f ec                   jg     c0142bb3 <invalidate_bdev+0x53>
c0142bc7:     45                      inc    %ebp
c0142bc8:     83 fd 02                cmp    $0x2,%ebp
c0142bcb:     7e bd                   jle    c0142b8a <invalidate_bdev+0x2a>
```

〔図8〕c0142bbaでの各レジスタ値を表示

```
$ grep c0142bba pebs |head
00000206 c0142bba 00000b00 ef11ed00 00000b00 f63c8000 0000a9e1 ef11ed80 00000000 f63c9e74
00000202 c0142bba 00000b00 ed775100 00000b00 f63c8000 00009657 ed775180 00000000 f63c9e74
00000202 c0142bba 00000b00 ebef9200 00000b00 f63c8000 000082ce ebef9280 00000000 f63c9e74
00000202 c0142bba 00000b00 eac00780 00000b00 f63c8000 00006f52 eac00800 00000000 f63c9e74
00000206 c0142bba 00000b00 e8ca6780 00000b00 f63c8000 00005bc9 e8ca6800 00000000 f63c9e74
00000202 c0142bba 00000b00 e73b2d80 00000b00 f63c8000 0000483b e73b2e00 00000000 f63c9e74
00000206 c0142bba 00000b00 e6238980 00000b00 f63c8000 000034be e6238a00 00000000 f63c9e74
00000202 c0142bba 00000b00 e4108b80 00000b00 f63c8000 00002134 e4108c00 00000000 f63c9e74
00000206 c0142bba 00000b00 e2842700 00000b00 f63c8000 00000da9 e2842780 00000000 f63c9e74
00000206 c0142bba 00000b00 eedf6100 00000b00 f63c8000 0000a77d eedf6180 00000000 f63c9e80

#eflags  liner_ip   eax      ebx      ecx      edx      esi      edi      ebp      esp
```

なことが行われているかは，オブジェクトを逆アセンブルすればよい．

```
$ objdump -S vmlinux
```

当該アドレス付近の逆アセンブル結果を**図7**に示す．そこで，c0142bbaでの各レジスタ値を表示する(**図8**)．左から4桁目がebxの値なので，アドレスef11ed00 ...でキャッシュミスが発生していることなどを容易に発見できる．

2.3　設計と実装

hardmeterの実装について解説する．ツールは次のコンポーネントからなる．

① メモリプロファイリング用ドライバ(Linuxカーネルへのパッチなど)

② ユーティリティ(ebs)

③ ユーザープログラム用API(Application Programming Interface)

性能モニタリング機能は，Linuxではデフォルトでは利用できないので，それを利用可能にするパッチが必要となる(**表2**)．

〔表2〕性能モニタリング機能用のパッチ例

名　前	開発者名	URL
perfctr	Mikael Pettersson	http://www.csd.uu.se/mikpe/linux/~perfctr/
brink_abyss	Brinkley Sprunt	http://www.eg.bucknell.edu/~bsprunt/
Rabbit	Don Heller	http://www.scl.ameslab.gov/Projects/Rabbit/
PAPI	Philip J. Mucchi, et. al	http://icl.cs.utk.edu/projects/papi/

〔表4〕perfctrのLinuxカーネルパッチとその追加機能

ファイル名	おもな追加機能
arch/i386/kernel/i8259.c	割り込みベクタ(perfctr_interrupt)の設定
arch/i386/kernel/irq.c	apic_lvtpc_irqsの設定
arch/i386/kernel/process.c	exit_thread()/copy_thread()/__switch_to()プロセス(スレッド)スイッチの処理
fs/proc/base.c	procファイルシステムへの追加
include/asm-i386/hw_irq.h	LOCAL_PERFCTR_VECTORの定義
include/asm-i386/processor.h	vperfctr構造体の定義とthread_structへの追加
kernel/timer.c	update_one_process()へperfctr_sample_thread()の追加

〔表5〕perfctrへの追加機能パッチ

ファイル名	ルーチン名	機　能
global.c	release_hardware gperfctr_init	perfctr_cpu_state_clean()を追加 perfctr_cpu_state_init()を追加
virtual.c	get_empty_vperfctr vperfctr_ihandler_signal vperfctr_ihandler_ibuffer sys_vperfctr_control vperfctr_read	perfctr_cpu_state_init()を追加 vperfctr_ihandler()から名称変更 新規 perfctr_cpu_set_ihandler()の設定 新規
x86.c	struct generic_setup_ibuffer generic_store_to_ibuffer generic_read_ibuffer p4_pebs_setup_ibuffer p4_pebs_read_ibuffer free_ibuffer set_ibuffer perfctr_cpu_state_init perfctr_cpu_state_clean perfctr_cpu_store_to_ibuffer perfctr_cpu_read_ibuffer	p4_ds_buffer構造体の定義 perfctr_ibuffer構造体の定義 perfctr_ibuffer_generic構造体の定義 perfctr_ibuffer_p4_pebs構造体の定義 新規 新規 新規 新規 新規 新規 新規 新規 新規 新規 新規
include/asm-i386/perfctr.h	struct	perfctr_cpu_stateの拡張 perfctr_p4_pebs_record構造体の定義
include/linux/perfctr.h		PERFCTR_FEATURE_PEBSの定義

● **メモリプロファイリング用ドライバの実装**

メモリプロファイリング用ドライバを，perfctr(**表3**)をベースに，PEBS対応の拡張を行って作成した(ソースコードはhttp://sourceforge.jp/projects/hardmeter/を参照のこと)．

perfctrは，標準のLinuxカーネルに対して**表4**のような拡張を行っている．

筆者らは**表5**のような機能追加を行った．

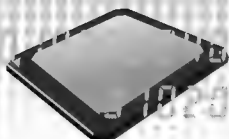
● **ユーティリティの実装**

ユーザーが簡単にメモリプロファイリングを行えるようなツール(ebsコマンド)を開発した．**表6**に，ebsのおもなファイルとその機能を示す

〔表3〕perfctrのおもなファイルとその機能

ファイル名	機　能
global.c	グローバルモードのドライバ
init.c	初期化ルーチン
virtual.c	プロセスごとのパフォーマンスカウンタ
virtual_stub.c	モジュール用スタブルーチン
x86.c	x86用パフォーマンスモニタリングカウンタ
x86_setup.c	x86用セットアップ

〔表6〕ebsのおもなファイルとその機能

ファイル名	機　能
ebs.c	メインルーチン
hardmeter.c	/proc/pid/perfctrドライバ
hardmeter.h	include file
p4_template.c	Pentium4の定義ファイル
rc.c	resource file (.hardmeterrc)の処理
self.c	高レベルAPIの実装

〔表7〕高レベルAPIと機能

API	機　能
int hardmeter_start(const char *name)	プロファイリング開始
int hardmeter_stop(void)	プロファイリング停止

〔表8〕低レベルAPIと機能

API	機　能
int hardmeter_init(const char **err)	初期化
const hardmeter_template_t *hardmeter_get_templates(const char**err)	テンプレート一覧の取得
const hardmeter_template_t *hardmeter_search_template(const char*name, const char **err)	テンプレートの検索
hardmeter_t *hardmeter_open(const hardmeter_option_t *opt, constchar **err)	/proc/PID/perfctrのオープン
hardmeter_t *hardmeter_attach_process(const hardmeter_option_t*opt, pid_t pid, const char **err)	pidにアタッチしてプロファイリング開始
hardmeter_t *hardmeter_start_process(const hardmeter_option_t*opt, const char *file, char *const argv[], const char **err)	プロセスをfork()してプロファイリング開始
int hardmeter_dump(hardmeter_t *h, const char *filename, int non_blocking, const char **err)	PEBSレコードの出力
int hardmeter_terminate(hardmeter_t *h, const char **err)	プロファイリング停止
int hardmeter_close(hardmeter_t *h, const char **err)	/proc/PID/perfctrのクローズ

〔図9〕$HOME/.hardmeterrcの記述例

```
# sample events in user-mode when non-zero. default 0.
resource_name.user     1
# sample events in kernel-mode when non-zero. default 0.
resource_name.kernel   0
# specify event name.
resource_name.event    l1_cache_miss
# output file
#   %Y  -  year (1970...)
#   %y  -  year (00..99)
#   %m  -  month (01..12)
#   %d  -  day of month (01..31)
#   %H  -  hour (00..23)
#   %M  -  minute (00..59)
#   %S  -  second (00..60)
#   %P  -  process id
resource_name.dumpfile /tmp/hardmeter.%y%m%d-%H%M%S.%P
# max sampling count. default 2000, max 524270 for pebs.
resource_name.count    200
# sampling interval. default 10000.
resource_name.interval 10000
```

● **API**

ユーザーアプリケーションから簡単に呼べるような高レベルAPIと，より詳細な機能を提供する低レベルAPIの2種類を実装した（**表7**，**表8**）．高レベルAPIでは，サンプリングの定義などをユーザーデフォルトディレクトリのリソースファイル（$HOME/.hardmeterrc）に定義しておき，アプリケーションプログラムはhardmeter_start(''resource_name'')とhardmeter_stop()でプロファイリングの開始，停止を行う．

リソースファイルの定義例：リソースファイルは，

リソース名.パラメータ名　パラメータの値

という書式で記述する．リソース名はスペースと‘.’以外の任意の文字を使える．resource_nameというリソース名でプロファイリングを行う場合は，$HOME/.hardmeterrcを**図9**のように記述する．

プロファイリングの定義はリソースファイル（$HOME/.hardmeterrc）にあるので，プログラムを変更しないでリソースファイルを変更するだけで簡単にプロファイリングができる．

低レベルAPIの場合，より細かい制御が可能になっている．hardmeterの元となったperfctrでは，測定するイベントごとに「どのパフォーマンスモニタリングカウンタを使用するのか，また，それぞれのカウンタの各ビットにどのような値を設定するのか」を明示的に指定する．これらの情報を指定するにはIntelのマニュアル[1]と首っぴきにどこにどのような値を入れる必要があるかを調べる必要がある．

hardmeterではユーザーの便を図って，これらのイベントをテンプレートとしてまとめて，テンプレートの名前を指定するだけで必要な情報を得られるようにした．hardmeterを利用する場合は，使用するテンプレートと若干のパラメータを指定するだけでイベントの測定ができるようにしてある．テンプレートの構造は，次のようになっている．

nameにテンプレートの名前，descriptionにその説明が入る．control，eventmaskにはテンプレートの実際の設定が入っている．is_pebsには，そのテンプレートがPrecise EBSなのか，Non-preciese EBSなのかの区別が入っている．ユーザープログラム側では，これらの値の参照のみを行う．nameのみが定義されていて他のメンバがNULLのエントリもあるが，これはebsコマンドが画面にイベント一覧を出力するときのコメントとして扱われる．

テンプレートの一覧を取得するときは，hardmeter_get_templates()を利用する．現在動いているマシンのCPUの種類に対応したテンプレートを戻すようになっている．現在の実装ではPentium4に対応したテンプレートのみを用意してあるが，将来は拡張する予定である（**図10**）．

テンプレートの名前でテンプレートを検索するときは，hardmeter_search_template()が利用できる（**図11**）．

おわりに

2回にわけて，メモリプロファイリングツールについて解説した．前編ではIA-32におけるハードウェアモニタリング機能について解説し，後編では，それを利用して実装したメモリプロファイリングツール(hardmeter)について解説した．

メモリプロファイリングツールを利用すると，簡単にアプリケーションおよびカーネルのメモリボトルネック(キャッシュミスなど)を発見できる．とくにPEBSによって精密にキャッシュミスなどのイベントが発生した時点での各レジスタ値などが取得できるので，より詳細な分析が可能となった．

今後の課題は，発見されたキャッシュミスなどをどうやって減少させるかというヒントを提示することなどが考えられる．現時点ではキャッシュミス多発地点は発見できても，どうやって，それを解決するかはプログラマまかせになっている．そこで，何らかのヒントを提示できるようにしたいと考えている．

今回開発したツールは，Webページ(http://sourceforge.jp/projects/hardmeter/)で公開しているので，読者諸氏の利用を待つ．メーリングリスト，掲示板などもあるので，ご意見，ご要望などをぜひ聞かせてほしい．このツールを，読者諸氏のアプリケーションの性能向上に役立てていただければ幸いである．

*　　　　　*

謝辞：メモリプロファイリングツールは平成14年度未踏ソフトウェア創造事業(プロジェクトマネージャー喜連川優東京大学教授)「OLTP性能向上を目的としたメモリプロファイリングツール」として支援を受け，開発を行った．

〔図10〕Pentium4に対応したテンプレート

```
struct hardmeter_template_t {
        const char *name;
        const char *description;
        const struct vperfctr_control *control;
        const hardmeter_event_mask_t *eventmask;
        int is_pebs;
};
typedef struct hardmeter_template_t hardmeter_template_t;
```

参考文献

1) Intel, The IA-32 Intel Architecture Software Developer's Manual Volume3: System Programming Guide, Order Number 245472, 2002
2) Dean, J. et. al, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-Of-Order Processors", Proceedings of Micro-30,December, 1997
3) Sprunt, B., "Pentium4 Performance Monitoring Features",IEEE Micro, July-August, 2002
4) 吉岡弘隆，平成14年度未踏ソフトウェア創造事業，「OLTP性能向上を目的としたメモリプロファイリングツール成果報告書」
5) 吉岡弘隆，「Intel系(IA-32)プロセッサのパフォーマンスモニタリングファシリティを利用したメモリプロファイリングツール」,『第44回プログラミングシンポジウム』，箱根，2003年
6) 吉岡弘隆，「OLTP性能向上を目的としたメモリプロファイリングツール」，『第14回データ工学ワークショップ』，電子情報通信学会，2003年
7) よしおかひろたか，「未踏ソフトウエア奮闘記」(前編/後編)，『日経ソフトウエア』，2003年7/8月号

よしおか・ひろたか　hyoshiok@miraclelinux.com
ミラクルリナックス(株)

〔図11〕hardmeter_search_template()の利用

```
int foo(....)
{
     hardmeter_option_t opt;
     const char *err;

     opt.template = hardmeter_search_template("l1_cache_miss", &err);
     opt.user = 1;
     opt.kernel = 0;
     opt.interval = 10000;
     opt.count = 2000;
     /* ... start sampling ... */
     hardmeter_handle = hardmeter_start_process(&opt, argv[0], argv, &err);
     /* ... */
}
```

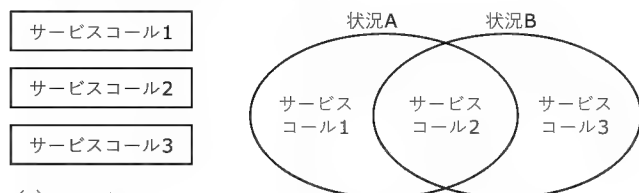
オープンソースのITRON仕様OS

TOPPERSで学ぶRTOS技術

第2回 シミュレータ環境を使って実際に動かしてみよう！

今井和彦

連載1回目をご覧になった方は「へぇ，TOPPERSって，フリーソフトウェアなんだ．試しにやってみようかな」と思われたことでしょう．リアルタイムOSに限らず，ソフトウェアに対する理解の早道は，本を読むだけでなく実際に動かしてみることです．幸い，TOPPERS/JSPカーネルにはシミュレーション環境が含まれており，実機がなくてもひととおりのことが試せます．今回は，これを用いて実際にプログラムを書いて，動かしてみましょう．

〔図1〕リアルタイムOSは難しい？

(a) サービスコール単体での見方　(b) 使われる場面とセットでの見方

(a)サービスコール単体だけ見てもOS全体は理解しにくいが，(b)使われるシチュエーションとセットでその背後にある考えも含めて考えると理解しやすい

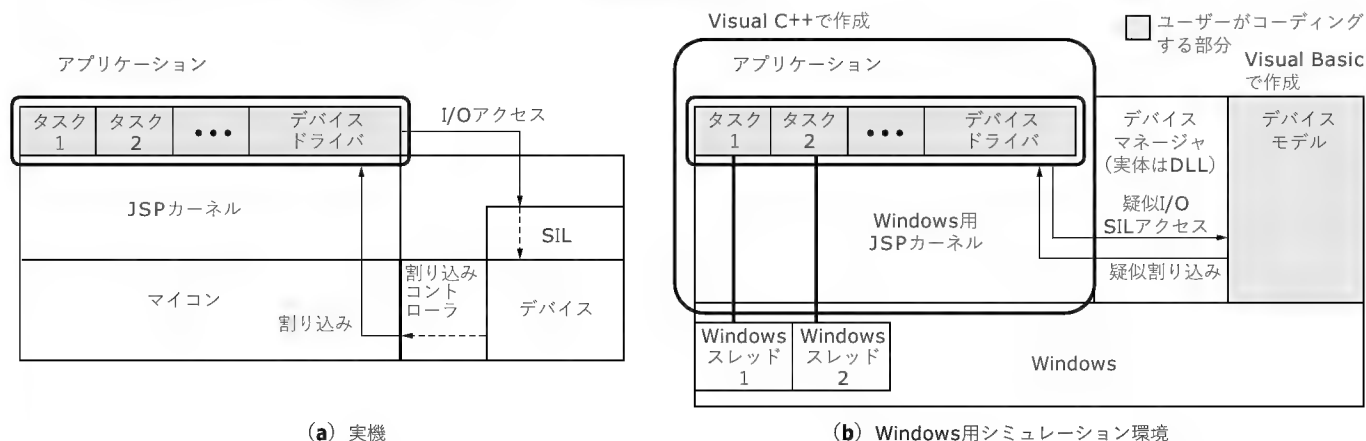
初心者がリアルタイムOSを学ぶ際のポイントはいくつかありますが，まず個々のサービスコール[注1]単体ではなく，その背後にある考え方を意識して，それが使われる状況とセットで理解することが大切です(**図1**)．同じサービスコールでも使う状況によっては意味が異なってくるのです(ちょうど，外国語を学ぶ際のイデオムや文脈に似ている)．それを知らないとμITRONの仕様書や参考書を読んでも「このサービスコールはどうやって使うのだろう？」，「どうして私が使いたい機能は用意されていないの？」といった疑問が残り，これではリアルタイムOSを導入しても開発効率アップどころではありません．

使用するツールとインストールの手順

TOPPERS/JSPカーネルのシミュレーション環境について述べることにします．JSPカーネルにはWindows用のシミュレーション環境が用意されており，実機がなくてもデバイスドライバを含めたシミュレーションもできるよう，自分でデバイスをモデリングできるしくみが提供されています(**図2**)．

本文中で扱う例題は，本誌発売時にはCQ出版社の本誌Web

〔図2〕TOPPERS/JSPカーネルの実機動作とシミュレーション環境の対応関係

アプリケーション
タスク1 | タスク2 | ・・・ | デバイスドライバ
I/Oアクセス
JSPカーネル
SIL
マイコン
割り込みコントローラ
割り込み
デバイス

(a) 実機

(b) Windows用シミュレーション環境

注1：一般には，OSのAPIはシステムコールと呼ばれているが，μITRON4.0仕様ではOSのAPIとソフトウェア部品のAPIを合わせてサービスコールと総称している．本稿でもこれにしたがう．

〔図3〕シミュレーション環境の実行画面

サイトからダウンロードできるようにする予定です注2ので，ぜひご自分のパソコン上で動かして体験してください．

開発は以下の三つのツールを用いて行います．

- TOPPERS/JSPカーネルWindows用シミュレータ（Windows 9xはサポート外）
- Visual C++ 6.0またはVisual C++ .NET
- Visual Basic 6.0

- インストール

Visual C++とVisual Basicはすでにインストールされているとします．

- JSPカーネルのコンフィギュレータとデバイスマネージャのビルド

Webサイトからダウンロードした`2003-10toppers.lzh`を解凍し，`jsp¥WINDOWS¥configure.vbs`スクリプトを実行します．Windows Scription Host実行環境がインストールされていない場合は，`jsp¥doc¥windows.txt`を参照してください．いくつかの質問に答えるとJSPカーネルのコンフィギュレータとデバイスマネージャがビルドされ，Visual C++を起動した後にサンプルのプロジェクトを開きます．このサンプルプログラムはJSPカーネルの簡単なデモプログラムになっています（**図3**）．興味のある方は`jsp¥WINDOWS¥sample1.c`のコメント部分に使い方の説明があるのでご覧ください．

デバイスシミュレーション環境で提供されているAPI一覧を**表1**，**表2**に示します．

表1のデバイスドライバ設計ガイドラインとは，その名のとおりデバイスドライバを設計する際の指針を示したもので，トロン協会により策定されています．このガイドラインでは，デバイスドライバの移植性を向上するため，デバイスへのリード/ライトにおけるエンディアンの違いを隠蔽する目的のAPI（システムインターフェースレイヤ）が規定されています．このインターフェースはWindows用シミュレーション環境と各種マイコン用JSPカーネルの両方で用意されているので，このインターフェースを介してデバイスにアクセスするようデバイスドライバを作っておけば，シミュレーション環境用にソースコードを書き換える必要がありません（Windowsシミュレーション環境と実機で共通のソースコードを利用できる）．

割り込み番号の割り当てを**表3**に示します．割り込み番号が大きいほど優先度が高く設定されており，タイマとシリアルはデフォルトで使用されています．割り込み要因の最大数は任意

〔表1〕アプリケーションから利用できるデバイスエミュレーション関連のAPI

デバイスドライバ設計ガイドライン互換インターフェース

`VB sil_reb_mem(VP mem)`
`VH sil_reh_mem(VP mem)`
`VW sil_rew_mem(VP mem)`
`void sil_rek_mem(VP mem, VP data, UINT len)`
指定されたサイズで，指定されたアドレスにマッピングされているデバイスからメモリ読み出しを行う．後述のDeviceReadのラッパ

`void sil_wrb_mem(VP mem, VB data)`
`void sil_wrh_mem(VP mem, VH data)`
`void sil_wrw_mem(VP mem, VW data)`
`void sil_wrk_mem(VP mem, VP data, UINT len)`
指定されたサイズで，指定されたアドレスにマッピングされているデバイスへメモリ書き込みを行う．後述のDeviceWriteのラッパ

独自インターフェース
`void InitializeComSupportModule(void)` デバイスエミュレーション関連の初期化を行う． 基本的にカーネルが起動時に初期化を行うため，ユーザーが呼ぶ必要はない
`void FinalizeComSupportModule(void)` デバイスエミュレーション関連の終了処理を行う． カーネル脱出時に終了処理を行っているので，能動的に呼ぶ必要はない
`int DeviceRead(unsigned long address, unsigned long size, void * storage)` デバイスからの読み込み address：デバイスを識別するためのアドレス値など（実メモリ空間とは独立） size：読み込む長さ storage：格納先へのポインタ 返り値：読み出されたデータのバイト数または－1（失敗） addressにマッピングされたデバイスからsizeバイトのデータを読み込み，storageに格納する．マッピングされたデバイスがない場合，関数は－1を返す．発行時にCPUはロック状態となる
`int DeviceWrite(unsigned long address, unsigned long size, void * storage)` デバイスへの書き込み address：デバイスを識別するためのアドレス値など（実メモリ空間とは独立） size：書き込む長さ storage：出力データを格納する領域へのポインタ 返り値：書き込まれたデータのバイト数または－1（失敗） addressにマッピングされたデバイスへstorageに格納されたsizeバイトのデータを書き込む，storageに格納する．マッピングされたデバイスがない場合，関数は－1を返す．発行時にCPUはロック状態となる

注2：http//www.cqpub.co.jp/interface/

〔表2〕デバイスモデル(Visual Basic側)から利用できるAPI
(TOPPERS/JSP on Windows Device Componentで提供される)

オブジェクト	DeviceControl
メソッド	Public Sub Connect() デバイスマネージャとの通信を確立させる. 確立前にConnect以外のメソッドを実行すると失敗する. 失敗するとE_FAILが返る
	Public Sub Close() デバイスマネージャとの通信を終了する. 失敗するとE_FAILが返る
	Public Sub RaiseInterrupt(ByVal inhno as long) カーネルプロセスにハンドラ番号inhnoの割り込みを発生させる. 失敗するとE_FAILが返る. 0以下の値は設定できない
	Public Sub Map(ByVal address as long, ByVal size as long) このデバイスをアドレスaddressからsizeバイトのサイズでマッピングする. 以後, 対象領域にアクセスがあると, イベントが発生する
	Public Sub Unmap(ByVal address as long) アドレスaddressを含むようなマップ済み領域のマッピングを解除する. 以後, 対象領域にアクセスがあってもイベントは発生しない
プロパティ	Public Valid as BOOL [R] このデバイスが有効であるときにはTrueとなる
	Public IsKernelStarted as BOOL [R] カーネルが起動しているとTrueとなる
	Public Offset as long [RW] 送受信データバッファのオフセット位置を指定/取得する. 単位はバイト単位
	Public AccessSize as short [RW] アクセス単位を指定/取得する. 数値は1, 2, 4のいずれかである
	Public Value as long [RW] プロパティOffsetの位置からプロパティAccessSizeバイトのリトルエンディアンにしたがい数値表現したものを取得/設定する. アクセスが行われるとOffsetの値をAccessSizeだけ増加させる. イベントOnRead時における読み込みは無効となる. イベントOnWrite時における書き込みは無効となる
イベント	Private Sub OnRead(ByVal address as long, ByVal sz as long) カーネルからaddress番地に対するszバイトの読み込みがあったことを通知する. デバイスはその返答となる値を格納しなければならない. イベント発生時にはOffsetは常に0となる. OnReadイベント発生時にはValueに対する読み込みはできない
	Private Sub OnWrite(ByVal address as long, ByVal sz as long) カーネルからaddress番地に対するszバイトの書き込みがあったことを通知する. デバイスはその値から適切な処理を行わなければならない. イベント発生時にはOffsetは常に0となる. OnWriteイベント発生時にはValueに対する書き込みはできない
	Private Sub OnKernelStart() カーネルが起動したことを通知する
	Private Sub OnKernelExit() カーネルが終了したことを通知する

注3:もちろん, ハードウェアも含めたシステム全体がそろって初めて実現可能な性質.

〔表3〕シミュレーション環境の割り込み番号の割り当て

優先度	割り込み番号	デバイス
低	1	空き
↑	2	
	3	シリアル
	4	タイマ
	5	空き
↓	6	
高	7	

〔リスト1〕タスク起動による情報通知

```
/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    iact_tsk(TASK_ID);  /* タスクを起動(READY状態へ) */
}

/*
 *  起動されるタスク
 */
void task(VP_INT exinf)
{
    proc_something();
    ext_tsk();       /* タスク終了(DORMANT状態へ) */
}
```

に設定できます(デフォルトでは7).

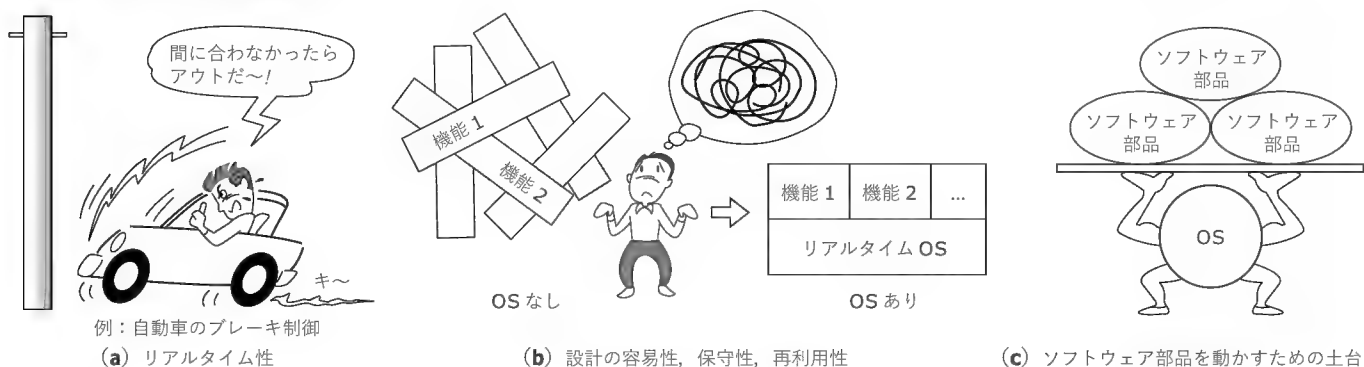
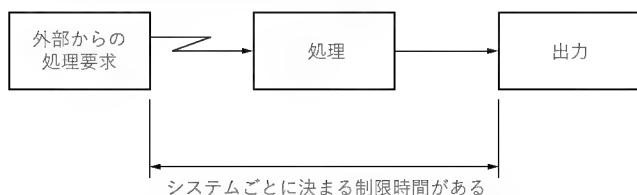
ここまででインストールは完了したと思います. 本記事で扱った例題は, 2003-10toppers.lzh内のex1～ex6ディレクトリに収録しています. たとえば, **リスト1**の「タスク起動による情報通知」の例題プログラムを実行するには, Visual Basicからプロジェクトファイルex1¥device¥Project1.vbpを開き, 実行します. 次にVisual C++からプロジェクトワークスペースex1¥vc_project¥toppers.dswを開き, ビルドを行って(アプリケーションプログラムを)実行します. なお, 本稿の例題はJSPカーネルRelease1.4の開発中のバージョンで動作確認を行いました. 正式リリース版では一部変更がある可能性があるので, ご了承ください.

ここで, JSPカーネルからはいったん離れ, リアルタイムOSの基本原理的な話題に戻ります.

リアルタイムOSを使う意義は?

図4に組み込みシステム用リアルタイムOSの役割を示します. 第1の役割は, システムの応答性を向上させることです. 組み込みシステムでは, 外界からの処理要求に対して制限時間内に応答する性質が重要視されます[注3]. この性質をリアルタイム性といいます. これはただ単に「数値計算の結果が正しい」という計算機の使い方とは次元の異なる話です. たとえば, 自動車のブレーキ制御では車輪がロックしていないかどうかを判断しながら制御を行っているわけですが, 結果が出力されるまで時間がかかりすぎて車体が障害物にぶつかってしまっては意

〔図4〕リアルタイムOSの役割

(a) リアルタイム性　(b) 設計の容易性，保守性，再利用性　(c) ソフトウェア部品を動かすための土台

〔図5〕リアルタイムシステムのモデル

味がありません．システムのリアルタイム性がいかに重要かわかるでしょう．

システムの規模が小さくてOSを使わなかった頃は，システムの応答性にかかわる部分を毎回，アプリケーションの一部として作り込んでいました．大雑把にいえば，この毎回作る部分を括りだして共通に使えるものとしてまとめたのがリアルタイムOSです．

リアルタイムOSの第2の役割は，設計の容易性，ソースコードの保守性，再利用性を向上させることです．これはリアルタイムOSがある場合に限った話ではないのですが，リアルタイムOSを使えば，システムを機能ごとに分割して保守性・再利用性に優れた設計をしやすくなります．これは製品のモデルチェンジを繰り返していく際に有利に働きます．

第3の役割は，前述の再利用性の利点をもう1歩押し進めた，ソフトウェア部品を動かす共通の土台としてのプラットホームとしての役割です．最近は組み込みシステムの大規模化・複雑化にともない，ソフトウェアの開発工数も急激に増え，システムのすべてを自社で開発することが困難になってきました．そのため，自社で蓄積したソフトウェア資産はもちろん，他社が作ったソフトウェア部品も取り入れることにより，開発期間を短縮することが多くなってきました[注4]．そうすると，リアルタイムOS単体の機能の優劣ではなく，その上で動くソフトウェア部品がいかに豊富にそろっているかが重要になってきました．

逆にいうと，ソフトウェア部品を提供するベンダはなるべく広く市場で用いられているOS用にソフトウェア部品を用意するほうがビジネス上有利になり，一度普及したOSはますます普及する傾向があります．

〔図6〕処理を複数個もったリアルタイムシステムのモデル

再考：そもそもリアルタイムシステムって何？

図5にリアルタイムシステムの典型的なモデルを示します．システムは外部からの処理要求を受けて必要な処理を行い，結果を出力します．このとき，システムごとに決まる制限時間内に結果を出さなければなりません[注5]．この締切のことをデッドラインといいます．

実際のシステムは複数の処理内容をもったものがほとんどですから，各処理の優先度の違いを考慮すると**図6**のようになります．この図だけ見ると，「リアルタイムOSがなくても，多重割り込みがあればリアルタイムシステムができるのでは？」と思われるかもしれません．各処理の内容が完全に独立であればそれでも良いのですが，システムが複雑になり各処理の内容が互いに絡み合ってくると，多重割り込みだけではさばき切れなくなってきます．具体的には**図7**の要因が挙げられます．**図7**の(a)～(c)の内容は以下のとおりです．

注4：とくにシステムにネットワーク接続機能をもたせるため，TCP/IPプロトコルスタックなどのソフトウェア部品が利用される例が増えてきた．
注5：すべての処理に制限時間があるわけではないが，ここではリアルタイムシステムの特徴を強調する意味で制限時間付きの処理を扱う．

コラム 1 サービスコールの返り値について

OSのサービスコールはエラー値を返すことがあります．μITRON4.0仕様では，サービスコールが正常に終了した場合はE_OK，異常終了時にはそれぞれのエラー要因ごとに定められた値を返すよう規定されています．本文中のサンプルプログラムでは，本質的な部分が見えにくくなるのでサービスコールの返り値をチェックしていませんが，実際のシステムに適用する際はきちんとチェックしましょう（**リストA**）．

また，ポーリング用のサービスコールや待ちの強制解除など，返り値が本質的な意味をもつ場合もあります．

〔リストA〕サービスコールの返り値のチェック

```
ER err;

err=service_call();
if(err==E_OK) {
    正常処理；
}else{
    異常処理；
}
```

(a) 優先度の異なる処理が同じ割り込みハンドラ内に混在する場合

一般に同じ割り込みをトリガにして行われる処理内容でも，本当に急いで処理しなければならない部分とそれほど急がない部分に分けられます．単純に一つの割り込みハンドラとして実装してしまうと，急がない部分の処理中に他の処理要求がブロックされてしまい，結果としてシステムの応答性が損なわれてしまいます．

(b) 排他制御

複数の割り込み(処理要求)が同じリソースを使う場合，最初の割り込みがそのリソースを使っている途中で，優先度の高い割り込みが入ったからといって単純に割り込みの優先度だけを見て処理を切り替えてしまうと処理の一貫性が保てなくなり，システムが破綻してしまいます．たとえば，**図7(b)**のように1台しかないプリンタを複数の処理要求で共有しているケースを考えれば，容易に想像がつくでしょう．これを防ぐには優先度とは別の指標で処理の切り替えを行う必要があります．これを制御の分野では「排他制御」と呼んでいます．

多重割り込みの中で排他制御を行うのは無理があるので，割り込みとは別の処理単位が必要になります．

(c) 処理の順序依存関係

組み込み制御に限った話ではありませんが，二つの処理が互いに独立ではなく，処理Aが終わらないと(必要なデータが揃わないなどの理由で)，処理Bが始められないということがよくあります．これを順序依存関係と呼び，割り込みだけではうまく表現できません．

タスクの導入

前節より，割り込みとは別の処理単位が必要なことがわかりました．μITRON4.0仕様ではその処理単位はタスクと呼んでいます[注6]．タスクに求められる性質は以下のとおりです(**図8**)．

(1) 割り込みハンドラより優先度が低い

(2) 優先度とは別の指標で処理を切り替えられ，待ち状態に入れる[注7]

(1)は**図7(a)**の割り込みハンドラの急がない部分の処理を受け持つことからくる要請です．タスクに処理を引き継いだ後は割り込みを受け付けることができます．タスク同士の処理の順番は優先度によって決まります(優先度ベーススケジューリング)．UNIXのプロセスのように各処理に対して平等にCPU時間を振り分けるのとは趣が異なります．

(2)は優先度以外の指標で処理を保留できるよう待ち状態を導入し，排他制御や順序依存関係をOSで扱えるようにするものです．

タスク状態

上記のような機能を実現するため，OSは状態遷移を用いてタスクを管理しています．μITRON4.0仕様のタスク状態を**図9**

〔図7〕割り込みだけでは対応できないケース

(a) 優先度の異なる処理が同じ割り込みハンドラ内に混在する場合

(b) 排他制御

(c) 処理の順序依存関係

注6：OSによってタスク，スレッド，プロセスと名称は異なる．

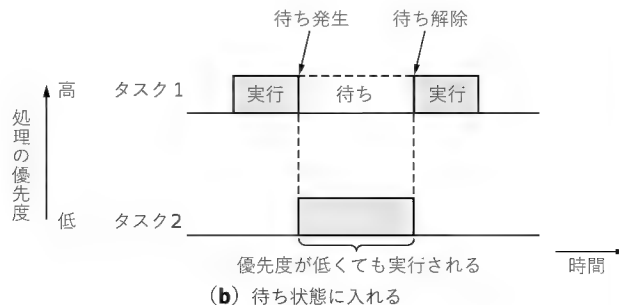
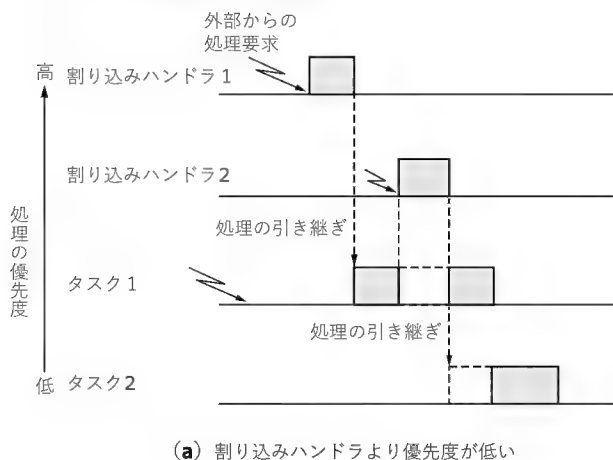
注7：アプリケーションによっては待ち状態は必要ない場合があるが，それはタスク間の依存関係がない場合．

〔図8〕タスクに求められる性質

(a) 割り込みハンドラより優先度が低い

待ち発生
待ち解除
高
タスク1
実行
待ち
実行
処理の優先度
低
タスク2
優先度が低くても実行される
時間

(b) 待ち状態に入れる

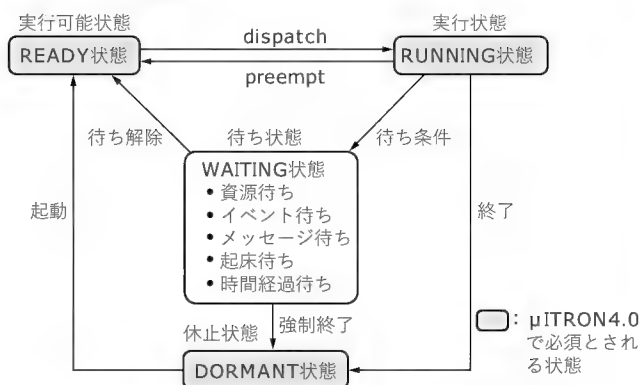
〔図9〕タスク状態

に示します注8．まずOSに登録されたタスクはDORMANT状態（休止状態）に入ります．DORMANT状態のタスクは起動されるとREADY状態（実行可能状態）に遷移します．実行可能なタスクのうち，もっとも優先度の高いタスクがRUNNING状態（実行状態）に選ばれ，実行されます．RUNNING状態とREADY状態の間の遷移は，優先度で決まります注9．

排他制御や順序依存関係を扱うため，READY状態とは別にWAITING状態（待ち状態）を用意し，優先度以外の要因で実行できないタスクを管理します．WAITING状態のタスクはそれぞれの待ち要因が解除されるとREADY状態へ遷移します．優先度が高いタスクも待ち要因が解除されるまでは実行されません．この待ち要因にはいくつかの種類があることがポイントで，これにより排他制御や順序依存関係など，さまざまなことを扱うことができます．待ち要因は，以下の5種類に分類されます．

- 資源待ち（セマフォ，メモリプール）
- イベント待ち（イベントフラグ）
- メッセージ待ち（メールボックス，データキュー）
- 時間経過待ち
- 起床待ち

割り込みハンドラからタスクへの情報通知

リアルタイムシステムでは外部からの処理要求のうち，急がない部分を割り込みハンドラからタスクへ引き継ぐ操作が非常に重要です（**図10**）．実際にOSの機能をどのように当てはめていけばよいのでしょうか．μITRONを例に見てみましょう．

μITRON4.0仕様では割り込みハンドラからタスクへの情報通知に適用できる機能として，

コラム2 VP_INT型について

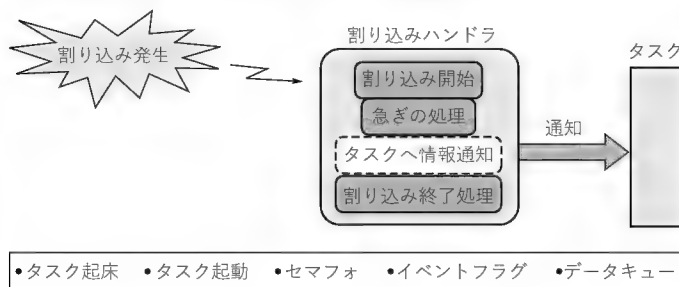
μITRON4.0仕様では，タスクの拡張情報（引き数）に使われているVP_INT型は「データタイプが定まらないものへのポインタまたはプロセッサに自然なサイズの符号付き整数」と規定されています．これは誤解を恐れず意訳すると「ポインタ型とint型のどちらでも格納できるサイズをもったデータ型注A」となります．実際のコーディングではタスクに渡したい引き数が1個のときはそのまま渡し，複数個のときはメモリ上にひとかたまりに配置して，タスクにはその先頭アドレス（＝ポインタ）を渡します．

この方法によって，OSのオーバヘッドを最小限にしつつ，引き数をたくさん渡したい場合にも対処しています．うまくできているものですね．

注A：プロセッサやコンパイラによっては，ポインタ型とint型のサイズが異なる場合がある．

注8：μITRON4.0仕様ではこの他にもタスク状態が定義されているが，ここまで理解できればかなり複雑なことも表現できる．
注9：現在実行中のタスクより優先度の高いタスクが実行可能になった場合は，その時点で直ちに実行するタスクを切り替える．このスケジューリング方式を優先度ベーススケジューリングという．

〔図10〕割り込みハンドラからタスクへの情報通知

•タスク起床　•タスク起動　•セマフォ　•イベントフラグ　•データキュー

- タスク起動
- タスク起床
- セマフォ
- イベントフラグ
- データキュー

の5種類が用意されています．一般に下のものほど通知できる情報量が多く，その代わりオーバヘッドが大きくなります．

● タスク起動

タスクをDORMANT状態からREADY状態へ遷移させて処理要求を通知します(**図11**，**リスト1**，p.162)．タスクは毎回終了し，待ち状態を使わないのでタスクのスタックを開放でき複数のタスクでスタック領域を共有する(メモリを節約する)実装も可能です．また，コンテキストを保存する必要がないので，汎用レジスタの数が多い最近のRISCマイコンでとくに有利です．通知できる情報量はイベント通知のみです．スタンダードプロファイルでは，1回まで処理要求をキューイングできます(後述)．

● タスク起床

タスクを起床待ちのWAITING状態からREADY状態へ遷移させて処理要求を通知します(**図12**，**リスト2**)．よく誤解される方がいるのですが，タスクを起床したからといって，タスクの数が増えるわけではありません．同一のタスクがスリープしたり起床したりするだけです(WAITING状態で止まっていたタスクが動き出すだけ)．通知できる情報量はイベント通知

コラム3　メールボックス機能の勘所

μITRON4.0仕様のメールボックス機能は，データの先頭アドレスだけが送られ，データ本体はコピーされない仕様になっています(**図A**，**図B**)．このことに注意しないと，簡単にデータ領域が破壊され，原因を見つけづらい不具合となります．ここでは初心者が陥りがちな落とし穴を2点，紹介します．

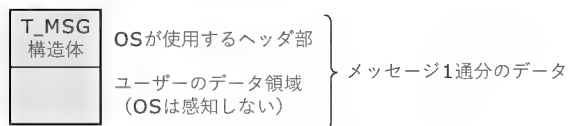
● ダブルポインタの扱いは慎重に

メッセージの先頭アドレスを受け取るタスクは**リストB**のⓐのように書ければ，シンプルでわかりやすいのですが，サービスコールの返り値は異常終了した際のエラー要因を通知するために使うので，それはできません．先頭アドレスを受け取る場所は**リストB**のⓑのように引き数へ移動させます．するとC言語では関数の引き数は値渡しなので，そのままの型を引き数に与えるだけでは所望の動作になりません．たとえば，**リストC**のようにint型のデータを受け取りたい場合は，int型データを書き込む領域の先頭アドレス，すなわちint *型のポインタ型を引き数として渡す必要があります．同様にT_MSG *型を受け取る場合の引き数はダブルポインタ型(T_MSG **型)になります．これはリアルタイムOSというより，C言語の文法レベルの問題ですが，初心者はよく混乱するようです．

● メモリプール機能との連携

メールボックス機能がメッセージの先頭アドレスしか送らないということは，言い換えれば，受信タスクがメッセージのメモリ領域を使い終わるまで，送信タスクはその領域を破壊してはいけないということです．メッセージを連続して送信する場合にはとくに注意が必要です(**図C**)．このあたりの制御をアプリケーション側で実装するのは結構大変ですが，μITRON4.0仕様にはメモリプール機能

〔図A〕メールボックス機能

〔図B〕メールのデータ構造

ヘッダ領域のデータ型としてT_MSG構造体が定義されている

〔リストB〕メッセージ受信時の先頭アドレスの受け取り方

```
ER err;
T_MSG*p;

p=rcv_mbx(MBX_ID);              ……ⓐ
err=rcv_mbx(MBX_ID, &p);        ……ⓑ
```

〔リストC〕C言語の関数呼び出しにおける結果の受け取り方

```
int data;

data=get_data();     ……ⓐ 戻り値として受け取る場合
get_data(&data);     ……ⓑ データを書き込む領域を引き数として与える
                          場合
```

のみです．スタンダードプロファイルでは，1回まで処理要求をキューイングできます（後述）．

● セマフォ

タスクを資源待ちのWAITING状態からREADY状態へ遷移させて処理要求を通知します（**図13**，**リスト3**）．通知できる情報量はイベント通知のみです．処理要求の最大キューイング数はセマフォカウンタの最大値として任意に設定できます．

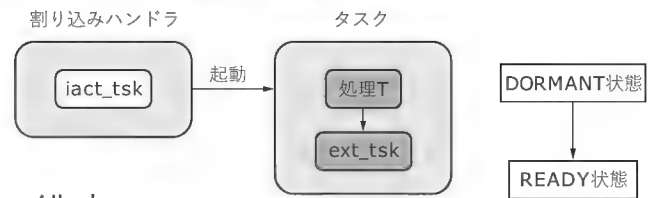
また，タスク側の処理〔**リスト3**中のproc_something()〕内に待ち状態には入るような操作が含まれていても，受け側のタスクを複数用意することである程度システムの応答性を確保できます．

本来，セマフォはタスク間の排他制御のためのしくみですが，このような使い方もできます．

● イベントフラグ

タスクをイベント待ちのWAITING状態からREADY状態へ遷移させて，処理要求を通知します〔**図14**，**図15**，**リスト4（a）**〕．処理要求は最大1回キューイングできます[注10]．イベントの有無だけでなく，ビットパターンを通知できるので，より多くの情報量を通知できます〔**リスト4（b）**〕．

また，この機能の大きな特徴として，複数のイベントを複合

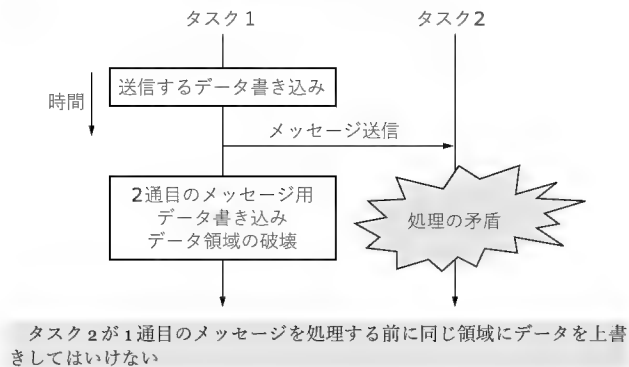
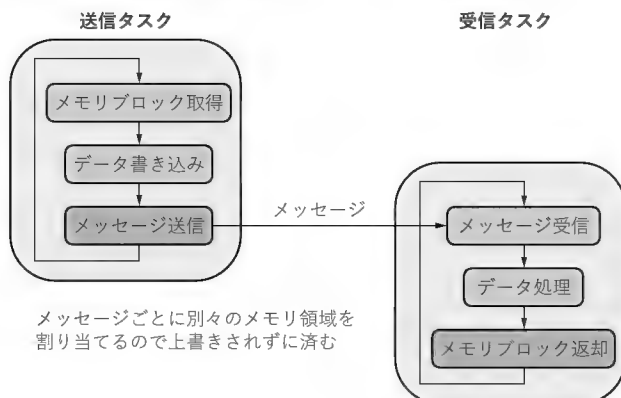
〔図11〕タスク起動による情報通知

メリット
- タスクスタックを開放できる
- コンテキストを保存する必要がない

備考
- 通知できるのはイベント発生のみ
  （それ以上の情報を通知するには共有メモリが必要）
- スタンダードプロファイルでは1回まで処理要求をキューイングできる

注10：フラグ生成時にフラグ属性としてTA_CLRを指定し，待ち解除と同時にイベントフラグの値をクリアする設定にした場合．

があるので，これを組み合わせると簡単に実現できます（**図D**，**リストD**）．

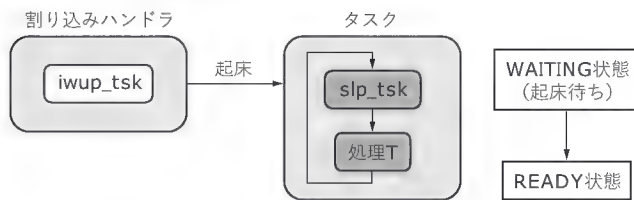
〔図C〕メールボックス機能で陥りやすいミス

タスク2が1通目のメッセージを処理する前に同じ領域にデータを上書きしてはいけない

〔図D〕メールボックス機能とメモリプール機能の連携

送信タスクではメッセージのデータサイズに見合ったメモリブロックを確保し，この領域に必要なデータを書き込んで送信します．受信タスクはメッセージを受信後，メッセージ内のデータを使い終わったら，メモリブロックをOSに返却します．

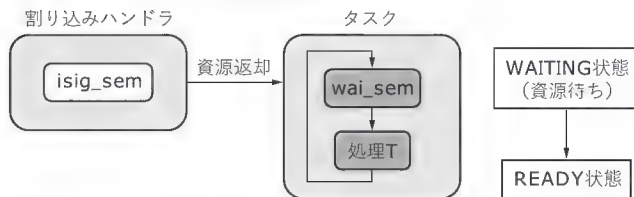
この手順でメッセージのメモリ領域が破壊される心配がなくなります．

〔リストD〕メモリプール機能との連携

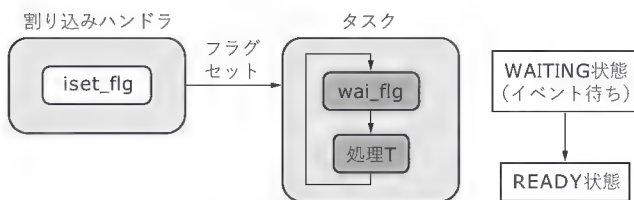
```
/*MESSAGEはユーザーが定義するメッセージ一つ分のデータ型 */
typedef struct{
    T_MSG header;   /* ヘッダ領域(OSが管理) */
    int data;       /* データ領域 */
}MESSAGE;

/*
 *  送信タスク
 */
void sender_task(VP_INT exinf)
{
    VP p;

    get_mpf(MPF_ID, &p);          /* メモリブロック確保 */

    /*pが指し示す領域にデータ書き込み */
    ((MESSAGE*)p)->data=xxxxxx;

    snd_mbx(MBX_ID,p);  /* メッセージ送信 */
}

/*
 *  受信タスク
 */
void receiver_task(VP_INT exinf)
{
    MESSAGE*p;

    rcv_mbx(MBX_ID, (T_MSG**)&p);   /* メッセージ受信 */

                /*pが指し示す領域のデータ使用 */
    proc_something(((MESSAGE*)p)->data);
    rel_mpf(MPF_ID, (VP)p);       /* メモリブロック返却 */
}
```

〔図12〕タスク起床による情報通知

メリット
- オーバヘッドが小さい

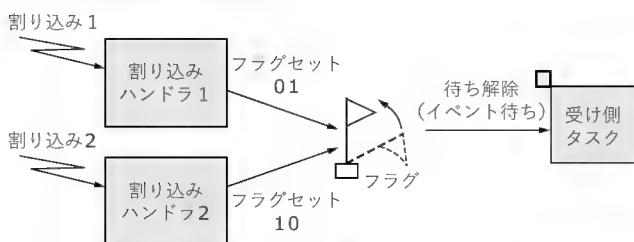
備考
- 通知できるのはイベント発生のみ
（それ以上の情報を通知するには共有メモリが必要）

〔図13〕セマフォによる情報通知

メリット
- 処理要求の最大キューイング数を調整できる
- 受け側のタスクを複数個用意できる

備考
- 通知できるのはイベント発生のみ
（それ以上の情報を通知するには共有メモリが必要）

〔図14〕イベントフラグによる情報通知（基本形）

メリット
- ビットパターンによって処理を選択できる
- 複数のイベントの複合通知ができる

備考
- 通知する情報量が多い場合，共有メモリが必要
- 要求をキューイングできない

〔図15〕複数イベントの通知

イベントフラグのセットは単なる上書きではなく，OSはその時点のフラグの値と論理和を取ってから代入する

複数のイベントに対して，ANDやORの待ち条件を設定することができる
→上書きしたら意味がない！

〔リスト2〕タスク起床による情報通知

```
/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    iwup_tsk(TASK_ID);  /* タスクを起床(READY状態へ) */
}

/*
 *  起床されるタスク
 */
void task(VP_INT exinf)
{
    while(1) {
        slp_tsk();  /* 起床待ち(WAITING状態へ) */
        proc_something();
    }
}
```

〔リスト3〕セマフォによる情報通知

```
/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    isig_sem(SAMAPHORE_ID); /* 資源返却 */
}

/*
 *  資源待ちのタスク
 */
void task(VP_INT exinf)
{
    while(1) {
        wai_sem(SAMAPHORE_ID);  /* 資源待ち(WAITING状態へ) */
        proc_something();
    }
}
```

させて待ち条件を設定することができます．複合条件としてはAND条件（すべてのイベントが発生した）またはOR条件（いずれかのイベントが発生した）の2種類が設定できます〔**リスト4(c)**〕．

● データキュー

タスクをデータ受信待ちのWAITING状態からREADY状態へ遷移させて処理要求を通知します（**図16**，**リスト5**）．処理要求の最大キューイング数はデータキューのサイズとして任意に設定できます．1ワードデータを使えるので，より多くの情報を通知できます．必要であれば割り込みハンドラ内でpsnd_dtq()を複数回実行して多くのデータを通知することも可能です．受け側のタスクを複数個用意できる点は，セマフォの場合と同じです[注11]．

表4(p.170)に示すように，それぞれの方法には特徴やトレードオフがあります．とくにイベントフラグは，複数のイベントの複合通知が簡単に実現できる面白い特徴があります．適材適所に使い分けましょう．

注11：受け側のタスクが複数でかつ，割り込みハンドラ内でpsnd_dtq()を複数回実行して2ワード以上送信する場合は，同一のタスクで連続してデータを受信できないので注意が必要．

〔リスト 4〕イベントフラグによる情報通知

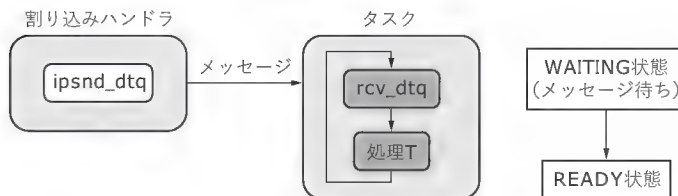
```
/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    iset_flg(FLAG_ID,0x1);  /* イベント通知 */
}

/*
 *  イベント待ちのタスク
 */
void task(VP_INT exinf)
{
    /* イベント発生時のフラグの値をコピーする領域 */
    FLGPTN flgptn;

    while(1) {
            /* イベント待ち(WAITING状態へ) */
        wai_flg(FLAG_ID,0x1,0x0, &flgptn);
        proc_something();
    }
}
```

(a) 基本形

```
/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    FLGPTN data;

    data=get_data();
    iset_flg(FLAG_ID,data|0x1);   /* イベント通知 */
}

/*
 *  イベント待ちのタスク
 */
void task(VP_INT exinf)
{
    /* イベント発生時のフラグの値をコピーする領域 */
    FLGPTN flgptn;

    while(1) {
            /* イベント待ち(WAITING状態へ) */
        wai_flg(FLAG_ID,0x1,0x0, &flgptn);
            /* イベントフラグのビットパターンをデータとして利用 */
        proc_something(flgptn);
    }
}
```

(b) 空きビットにデータをのせる場合

```
/*
 *  割り込みハンドラ(イベント1を通知)
 */
void interrupt_handler1(void)
{
    iset_flg(FLAG_ID,0x1);  /* イベント1を通知 */
}

/*
 *  割り込みハンドラ(イベント2を通知)
 */
void interrupt_handler2(void)
{
    iset_flg(FLAG_ID,0x2);  /* イベント2を通知 */
}

/*
 *  イベント待ちのタスク
 */
void task(VP_INT exinf)
{
    /* イベント発生時のフラグの値をコピーする領域 */
    FLGPTN flgptn;

    while(1) {
            /* イベント1とイベント2のAND待ち */
            /* (両方のイベントが発生したら待ち解除) */
        wai_flg(FLAG_ID,0x3,TWF_ANDW, &flgptn);
        proc_something();
    }
}
```

(c) 複数イベントの複合条件

〔図 16〕データキューによる情報通知

特徴
- 1ワードデータを通知できる
- 要求をキューイングできる最大数はデータキューのサイズで決まる
(スタンダードプロファイルでも2回以上キューイングできる)

〔リスト 5〕データキューによる情報通知

```
/*
 *  割り込みハンドラ
 */
void interrupt_handler(void)
{
    ipsnd_dtq(DTQ_ID,0x1);   /* イベント通知 */
}

/*
 *  データ受信待ちのタスク
 */
void task(VP_INT exinf)
{
    VP_INT data;    /* データ受信時のデータをコピーする領域 */

    while(1) {
            /* データ受信待ち(WAITING状態へ) */
        rcv_dtq(DTQ_ID, &data);
        proc_something(data);
    }
}
```

起動要求のキューイング

ところでJSPカーネルが準拠しているμITRON4.0仕様スタンダードプロファイルでは「タスクの起動要求や起床要求は1回以上キューイングできること」と規定されているのですが，これを読んだことのある方は，この機能がどうして必要なのか疑問に思われたかもしれません．この機能は「他の実行単位から処理要求を受けている」という観点で見たときに初めて意味が出てくるのです(**図17**，p.171)．

1回目の処理が完了しないうちに2回目の処理要求が入った場合，2回目の割り込みハンドラから発行されるタスク起動要求/起床要求はキューイングされます．1回目の処理が完了した時点でタスクが終了/スリープしようとしても，キューイングされていた起動要求/起床要求とキャンセルされ，タスクは状態遷移しません．結果として，2回目の処理要求も取りこぼさないで処理されます．

〔表4〕各種通知方式の比較

方　式	サービスコール	情報量	オーバヘッド	処理要求の最大キューイング数	受け側のタスク数	備　考
タスク起動	ext_tsk iact_tsk	少ない	小	1回※1	単数	タスクスタックを開放できる． コンテキストを保存する必要がない
タスク起床	slp_tsk iwup_tsk	少ない	小	1回※1	単数	
セマフォ	wai_sem isig_sem	少ない	中	調整可能	複数も可能	
イベントフラグ	wai_flg iset_flg	中	中	1回※2	単数※3	ビットパターンによって処理を選択できる． 複数イベントの複合条件を設定できる
データキュー	rcv_dtq ipsnd_dtq	多い	大	調整可能	複数も可能	

※1：スタンダードプロファイルでは1回以上と規定されている（アプリケーション側で仮定して良いのは1回まで）．
※2：イベントフラグ生成時に属性として，TA_CLRを指定し，タスクの待ち解除時にイベントフラグのすべてのビットをクリアする設定にした場合．
※3：スタンダードプロファイルでは複数のタスクが同一のイベントフラグを待つことはサポートしなくてもよい．
一般に通知できる情報量とオーバヘッドはトレードオフの関係にある．状況に応じて適所適材に使い分けることが重要．

コラム4　周期起動タスクについて

タスクの周期起動を行いたい場合，どのようなプログラムにすればいいのでしょうか．μITRONでは，タスクを登録する際にその属性として「周期起動」を設定して周期起動タスクを直接扱うような機能は用意されていません．その代わり，「周期ハンドラ」という機能があり，登録した関数を設定した時間間隔でカーネルが呼び出してくれます．

● 周期ハンドラとは

周期ハンドラは非タスクコンテキストで実行されます．つまり，周期ハンドラはタスクより割り込みハンドラに近い扱いです（実際，周期ハンドラもタイマ割り込みハンドラの一種と考えれば，わかり易いかと思う）．

手順を以下に示します．

1）静的API CRE_CYC()で周期ハンドラをカーネルに登録

起動間隔はここで設定します．

コンフィギュレーションファイル：

```
CRE_CYC(CYCHDR1, {TA_HLNG|TA_STA,0,
                  cyclic_handler,PERIOD_TIME,0});
```

TA_HLNGとTA_STAはμITRON4.0仕様で規定される属性値で，TA_HLNGは周期ハンドラが高級言語（C言語）で記述されていることを示します．また，属性値としてTA_STAを指定すると，周期ハンドラ生成後に周期ハンドラを動作している状態にします（最初の周期のカウントダウンを開始する）．

2）周期ハンドラ内の処理

2-1 処理が短い場合

→周期ハンドラ内ですべて処理してしまう

周期ハンドラ：

```
void cychdr(VP_INT exinf)
{
  周期的に行いたい処理;
}
```

2-2 処理が長い場合

→周期ハンドラはタスクに通知するのみに留め，
　実際の処理はタスクにまかせる

周期ハンドラ：

```
void cyclic_handler(VP_INT exinf)
{
    iwup_tsk(TASK_ID);    /* タスク起床 */
}
```

通知される側のタスク：

```
void task(VP_INT exinf)
{
        while(1) {
                slp_tsk();    /* 起床待ち */
                周期的に行いたい処理;
        }
}
```

周期ハンドラからタスクへの情報通知をどのように行うかは，割り込みハンドラからの通知の場合とまったく同じ議論が適用できるので，本文の解説を参照してください．

周期ハンドラには引き数が一つ渡せます．タスクIDを引き数として渡すと周期ハンドラを使い回すことができて，起動周期が異なる処理が複数あってもタスクごとに周期ハンドラを用意する必要がないので，メモリの節約になります（**リストE**）．

〔リストE〕複数のタスクでの周期ハンドラの共有

```
/* コンフィギュレーションファイル：
   同じ周期ハンドラに対して引き数と周期を変えて２回登録する */
CRE_CYC(CYCHDR1, {TA_HLNG|TA_STA, (VP_INT)TASK_ID1,
                  cyclic_handler,PERIOD_TIME1,0});
CRE_CYC(CYCHDR2, {TA_HLNG|TA_STA, (VP_INT)TASK_ID2,
                  cyclic_handler,PERIOD_TIME2,0});

/*
 *  周期ハンドラ
 */
void cyclic_handler(VP_INT task_id)
{
    iwup_tsk((ID)task_id);  /* タスク起床 */
}

/*
 *  起床されるタスク１
 */
void task1(VP_INT exinf)
{
    while(1) {
        slp_tsk();          /* 起床待ち */
        proc_something();   /* 周期的に行いたい処理 */
    }
}

/*
 *  もう一つのタスクも同様
 */
```

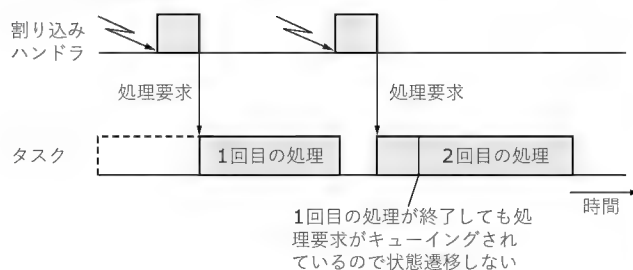
おわりに

今回は，システムの応答性という観点で，割り込みからの情報通知を中心にリアルタイムOSの使い方を解説しました．

初心者の方もリアルタイムOSの見た目の機能の多さに惑わされずに，必要な機能から一つずつ覚えていけば大丈夫です．本稿がその一助になれば幸いです．

〔図17〕処理要求のキューイング

参考文献

1）デバイスドライバ設計ガイドライン
http://www.assoc.tron.org/itron/home-j.html
2）『μITRON4.0ガイドブック』，パーソナルメディア，2001年
3）TOPPERSプロジェクト ホームページ
http://www.ertl.jp/TOPPERS/
4）μITRON4.0仕様
http://www.itron.gr.jp/SPEC/mitron4-j.html
5）『リアルタイムOSと組み込み技術の基礎』，CQ出版，2003年

いまい・かずひこ　宮城県産業技術総合センター

コラム5 組み込みソフトウェアにはmain()関数がない？

C言語の参考書を見ると最初に実行される関数はmain()関数と相場が決まっていますが，本稿のサンプルプログラムにはmain()関数が一つも出てきません．μITRONのような組み込み用OSでは一般にmain()の呼び出しは行っていません[注B]．組み込みソフトウェアでは電源投入時にスタートアップルーチンと呼ばれるソフトウェアが起動され，

- プロセッサの初期化：動作モード設定，スタックポインタの設定
- ボード（システム）の初期化：とくにメモリ周りの初期化

が行われた後，

- カーネルの初期化：タスクなどのカーネルオブジェクトの初期化
- スケジューリング開始：適切なタスクを選択して実行

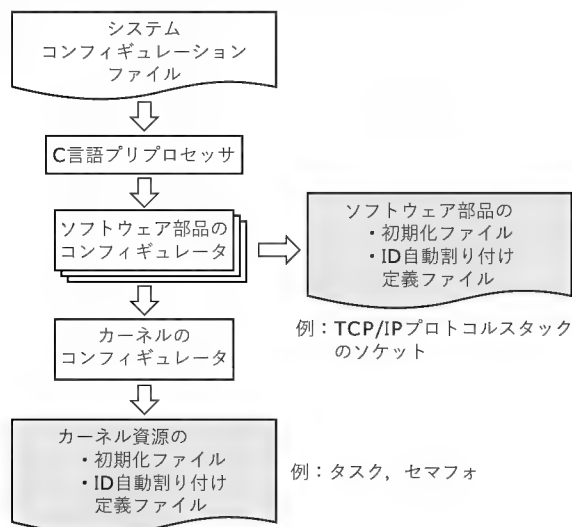
と進みます．

ところでタスクをカーネルに登録する部分は何が行っているのでしょうか？

μITRON4.0仕様スタンダードプロファイルではタスクなどのカーネルオブジェクトは静的に（コンパイル時に）生成するように規定されています．この作業はカーネルの内部構造に大きく依存するのでアプリケーション開発者にこれを直接やってもらうのでは非常に煩雑ですし，ソフトウェア部品の移植性という観点からも好ましくありません．

そこでμITRON4.0仕様ではコンフィギュレータというツールが導入されました（**図E**）．アプリケーション開発者は静的APIと呼ばれる専用のAPI[注C]を使って，必要なカーネルオブジェクトをコンフィギュレーションファイルに記述しておきます．コンフィギュレータはコンフィギュレーションファイルを読み込んで，カーネルの内部構造に合わせたC言語ソースコード（おもに自動割り当てされたIDの定義とカーネル内部で使用するデータテーブル）に変換します．コンフィギュレータはカーネル開発者によりアプリケーション開発者に提供されます．一度コンフィギュレーションファイルを作成しておけば，他のμITRON4.0仕様OSでもそのまま使えるので，アプリケーション開発者やソフトウェア部品開発者の負担が軽減されます．また，プログラムの変更でカーネルオブジェクトの数が増減しても，タスクなどの各カーネルオブジェクトのIDはコンフィギュレータが自動的に割り振ってくれるので，ケアレスミスを防ぐことができます．これは，外部で開発されたソフトウェア部品を導入する際にとくに効いてきます．

興味のある読者は，JSPカーネルのサンプルプログラムに含まれるコンフィギュレーションファイルsample1.cfgをご覧ください（コンフィギュレーションファイルの拡張子はcfgとする約束になっている）．

〔図E〕システムコンフィギュレーションの手順

注B：汎用機向けプログラマからの転向者を考慮して，main()関数の呼び出しを行っている流儀の組み込み用OSもある．

注C：文法的には，通常のカーネルオブジェクト生成用のサービスコールを大文字に置き換えたものを使用するだけなので，アプリケーション開発者が新たに覚えなければならない事項はほとんどない．たとえば，タスク生成のサービスコールcre_tsk()に対応する静的APIはCRE_TSK()で，引き数の並びもまったく同じ．

PC/ATのさまざまな資源を管理する

ACPIによるPC/ATの電力管理とコンフィグレーション

安達健一

Advanced Configuration and Power Interface

従来PC/ATの電力管理にはAPM(Advanced Power Management)が使われてきた．しかし昨今のハードウェアの進化によって，APMでは要求に応じられないような局面も出てきた．そこで登場したACPI(Advanced Configuration and Power Interface)は，電力管理はもちろんのこと，多様な資源のコンフィグレーションを可能にしている．
後編の今回は，LinuxとWindowsについて，ACPIの実際の挙動をカーネルデバッガを用いて追いかけてみる．
(編集部)

前編では，まずACPIの基本コンセプトを確認し，主要なACPIテーブルと，システム初期化時におけるACPIの役割を解説しました．

今回は「OSから見える」イベントとして通知されたACPIイベントを，OSがどのようにハンドルしているのか，実例を中心に解説したいと思います．

ACPIイベントのハンドル――Linux編

それでは，ここからは読者の皆様も手を動かして，OSがACPIイベントをハンドルするようすをカーネルデバッガで追いかけてみましょう．

● カーネルデバッガ

オープンソースで納得の行く追跡がしたいという向きのために，Linuxでの実装を中心にOSのACPIイベントハンドリングを見ていくことにします．手っ取り早く実機をいじりたいので，ターゲットマシン単体でカーネルデバッグができるKDB(Built-in Kernel Debugger)を用います．

LinuxのACPIサポートは，現在カーネル2.5系をメインターゲットに，非常に活発に変更が行われています．本来であれば，2.5系で進めたいところですが，KDBが間に合っている2.4系で追っていくことにします．

また，Windowsに関しても，最近は製造元である米国Microsoftが高品質なデバッガパッケージを無償提供し，デバッグに必要なシンボルも公開しています．そこで，これらを利用したWindows XPでのACPIイベントハンドリングのようすも一通り見ておきます．

● LinuxカーネルへのKDBの適用

Linux標準カーネルを本家のアーカイブサイト(`http://www.kernel.org/`)などから入手します．今回は，バージョン2.4.20を対象とします．

続いて，Linuxカーネルのバージョンに合致するKDBをSGIのサイト(`http://oss.sgi.com/projects/kdb/`)からダウンロードしてください．KDBはカーネル本体へのパッチとして提供されるので，解凍して次のように適用してください．

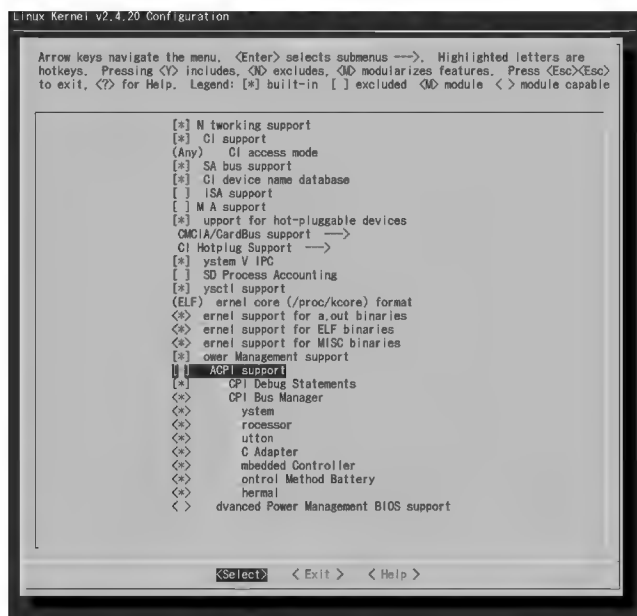
```
# cd linux-2.4.20
# patch -p1 < kdb-v4.2-2.4.20-common-1
# patch -p1 < kdb-v4.2-2.4.20-i386-1
```

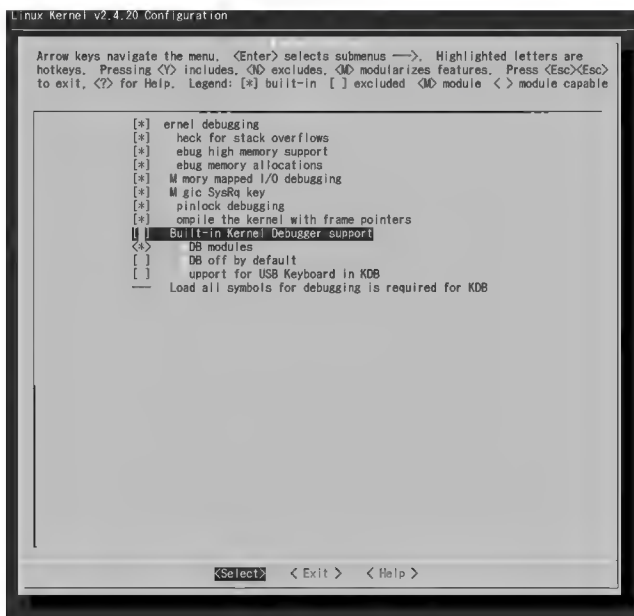
次にACPIサポートとKDBを有効にするため，カーネルを再構築します．たとえばカーネルのコンフィグレーションを，

```
# make menuconfig
```

で行う場合は，次の箇所で設定します．

```
Main Menu
  General setup--->
      [*]Power Management support
      [*]   ACPI support
```
以下，ACPIサポートのサブオプションは必要に応じて有効化(**図1**)
```
  Kernel hacking--->
      [*]Kernel debugging
      [*]   Built-in Kernel Debugger support
```
以下，カーネルデバッグ関連のサブオプションは必要に応じて有効化(**図2**)

〔図1〕カーネルコンフィグレーションの画面

〔図2〕カーネルデバッガを有効にする

〔図3〕ACPIイベントの取り扱い

なお，カーネルコンフィグレーションメニューの配置はバージョンごとに変化していますから，2.4.20以外でテストされる方は注意してください．

● 再構築済みカーネルのブート

はじめにKDBがオンになっているか，確認します．

```
# cat/proc/sys/kernel/kdb
1
```

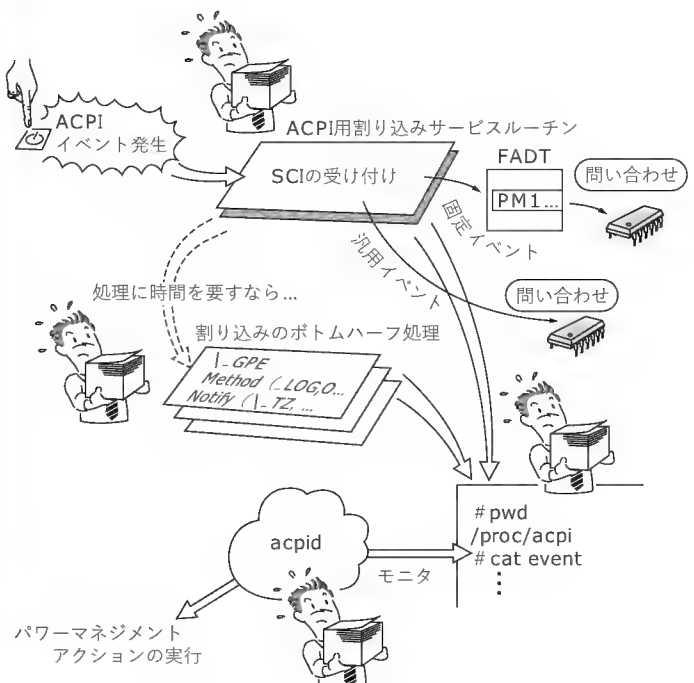
1が返ってくれば，KDBはオンになっています．

ACPIサポートを有効にしたカーネルでブートすると，/procの下にacpiというディレクトリが作られます．この下にeventというエントリが現れます．Linuxでは，電源ボタン押し下げのようなACPI管理下のイベントを拾うと，このエントリにイベント内容が出力されます．

catコマンドでこのエントリを監視しておくと，ACPIイベントが発生したときに，対象デバイスとイベントの種別が表示されます．

```
# cat/proc/acpi/event
button power 00000080 00000000
ac_adapter 0 00000080 00000000
battery 0 00000080 00000000
  ……
```

したがって，ユーザーランドのプログラムで，たとえば「ACPIイベントが上がったときにシステムスリープに入る」といったアクションを起動したい場合は，このエントリをモニタしておけばよいわけです．

そのもっともシンプルな例が，acpidというACPIイベントを監視してアクションを起こすデーモンです．acpidは現在，SourceForge上にACPIメインとは別のプロジェクト（http://acpid.sourceforge.net/）として存在し，RPMパッケージも配布されています．

今回は，ACPIサブシステムにACPIイベントの発生が通知されてから，この/proc/acpi/eventエントリに情報を出力するまでの処理を追いかけます（**図3**）．

● ソースコードの机上チェック

動作中のシステムをKDBで覗く前に，ソースコード上でLinuxのACPIイベントハンドリングを確認しておきましょう．

なお，繰り返しになりますが，Linux上のACPIサポートは，現在もっともアクティブに開発が進められている分野の一つです．そのため，ここで動作を調べているバージョン2.4.20と最新のものとは大幅に内容が変更されています．

ただし，本稿では対象をACPIイベントハンドリングの大枠だけに限定しており，この根本的な処理範囲では，2.4.20当時から現在に至るまで本質的な変化はそう多くはありません．

ソースコードのナビゲーションは，読者の皆様がふだん使い慣れている環境で行ってください．筆者は，WebベースでクロスリファレンスができるLinux Cross-Referencing projectのサイト（http://lxr.linux.no/source/）やRed HatのSource Navigator（**図4**）を使っています．

▶ SCI受け付け

ACPIのコードはlinux-2.4.20/drivers/acpi/にあります．ソースコードのロケーションは，断りがない限りLinux 2.4.20標準カーネル上での位置を，drivers/acpi/からの相

対パスで表記しています．

前編で強調したように，ACPIのキーコンセプトの一つが，「OSから見えない」SMI(System Management Interrupt)から「OSから見える」SCI(System Control Interrupt)へのパラダイムシフトでした．

したがって，ACPIではプラットホームで発生するACPI管理下のイベントは，OSの用意する関数によって明示的に処理を行わなければなりません．

このSCIを受け付ける割り込みサービスルーチンがevents/evsci.cのacpi_ev_sci_handler()(**リスト1**)です．この関数の役目はきわめて明瞭で，本当にACPIイベントが上がっていることを確認のうえ，イベント種別を判定して処理を振り分けています．

```
+acpi_ev_sci_handler()              // ……①
  ……
  +acpi_ev_fixed_event_detect()     // ……②
  +acpi_ev_gpe_detect()             // ……③
```

▶イベント種別の判定

ACPIイベントには，どのACPI対応マシンにも共通して存在するような，固定イベント(Fixed Feature Event)と，汎用的なイベント〔General Purpose Event(GPE)〕があります．

前者は，イベント発生が通知されるレジスタの名称やビット位置が，ACPI仕様書で固定されていて，イベントのハンドルもACPI OSの中でほぼ完結しています．上の例では②のevents/evevent.c:acpi_ev_fixed_event_detect()が，この固定イベントの検出と処理を担っています．

後者の汎用イベントについて解説します．どのレジスタのどのビットをどのイベントに割り振るかは，マシンの製造元が決定し，かつイベント処理用のコードも，GPEハンドラと総称されるコントロールメソッドとしてACPI BIOS内に用意されます．ACPI OSは汎用イベントを受け付けた場合，ACPI BIOSのGPEハンドラをAMLインタプリタを介して実行し，当該ACPIイベントが期待する処理を行います．上では③のevents/evevent.c:acpi_ev_gpe_detect()が，この汎用イベントの検出と処理を担っています．

では，さっそく①～③の関数にブレークポイントを設定したうえで，電源ボタン押し下げというACPIイベントを起こし，OS側の動作を追跡しましょう．

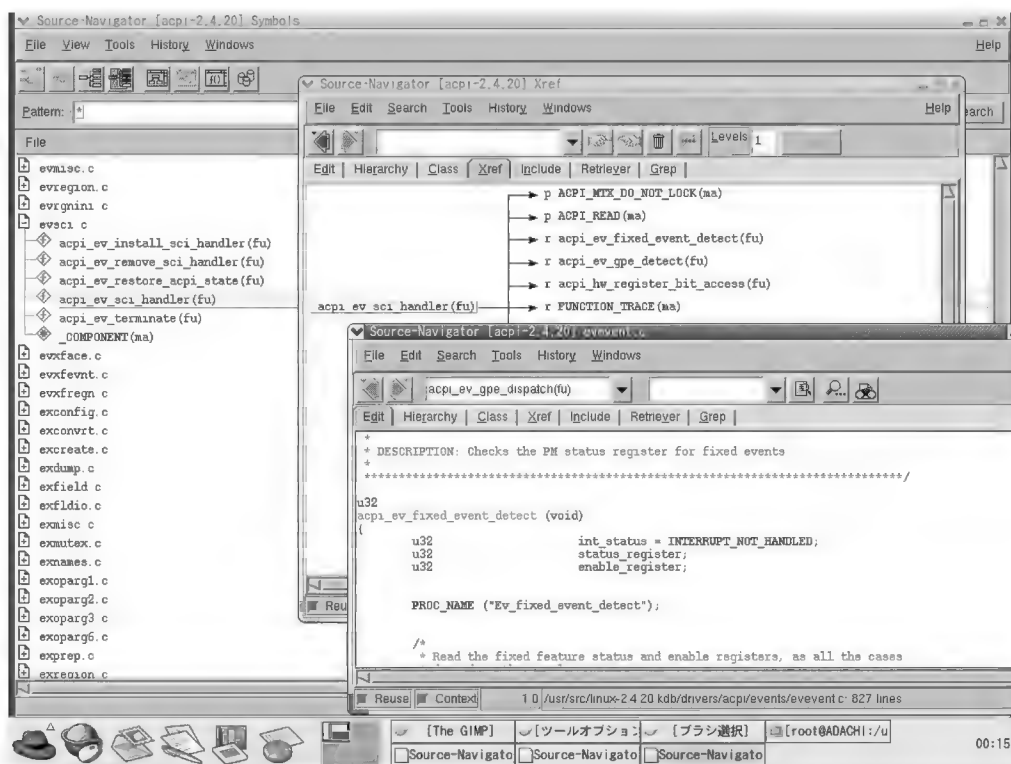
KDBでは，Pauseキー押し下げなどで，デバッガ画面に制御を渡します．“[CPU番号]kdb>”というプロンプトが現れたら，bpコマンドでブレークポイントをセットし，goコマンドでいったんデバッガから抜けます(**図5**)．

電源ボタンを押し下げると，ブレークポイント0のSCI割り込みサービスルーチンでブレークします(**図6**)．

ここで，goコマンドで進めると，残りのブレークポイントにも順次引っ掛かるのがわかります．

ところで，今追いかけている処理は，SCI受け付け後，それが固定イベントか，汎用イベントかを識別することでした．もう一度関数の呼び出し関係を整理しておきましょう．

```
+acpi_ev_sci_handler()              // ……①
```

〔図4〕Source Navigatorの画面

……
    +acpi_ev_fixed_event_detect() // ……②
    +acpi_ev_gpe_detect()   // ……③

②で，固定イベントが上がっていないかチェックしています．固定イベントであれば，ACPI仕様書に規定されたPower Managementイベントブロック(PM1a_EVT_BLK，PM1b_EVT_BLK)のステータスレジスタで，発生したイベントのビットがセットされ，かつイネーブルレジスタの対応するビットもセットされているはずです．

Power Managementイベントブロックは，実際にはチップセット内に配置されたレジスタで，通常のI/Oポートと同様にアクセスできる領域です．また，そのアドレスはFADTテーブルから辿れることは前編で述べたとおりです．

この先，②は次のように進んでいきます．

```
+acpi_ev_fixed_event_detect()
  +acpi_hw_register_read()
    +acpi_hw_low_level_read()
      +acpi_os_read_port()
```

最後のacpi_os_read_port()が含まれるos.cには，ホストOSであるLinuxが，ACPI CAサブシステムの要求に応じてハードウェアへ直接アクセスする関数群が定義されています．この中にはおなじみのinb/inw/inl/outb/outw/outlやメモリ，PCIコンフィグレーションレジスタへアクセスするコードが並んでいることがわかると思います(**リスト2**)．

リスト3に，I/Oポートから値を読み出すまでの処理のうち，本文に書ききれなかった部分のデバッグログを示します．

▶イベントハンドリング

SCIは別の見方をすれば，ACPI"デバイス"からの割り込みです．そのため，一般の割り込みサービスルーチン同様，速度的な制約を受け，SCIハンドルに時間を要する場合は，その全

〔リスト1〕acpi_ev_sci_handler()

```
/*******************************************************************
 *
 * FUNCTION:    Acpi_ev_sci_handler
 *
 * PARAMETERS:  Context   - Calling Context
 *
 * RETURN:      Status code indicates whether interrupt
 *                                                   was handled.
 *
 * DESCRIPTION: Interrupt handler that will figure out what
 *              function or control method to call to deal with
 *              a SCI.  Installed using BU interrupt support.
 *
 ******************************************************************/

static u32
acpi_ev_sci_handler (void *context)
{
    u32       interrupt_handled = INTERRUPT_NOT_HANDLED;


    FUNCTION_TRACE("Ev_sci_handler");


    /*
     * Make sure that ACPI is enabled by checking SCI_EN.
     * Note that we are required to treat the SCI interrupt as
     * sharable, level, active low.
     */
    if (!acpi_hw_register_bit_access (ACPI_READ,
                          ACPI_MTX_DO_NOT_LOCK, SCI_EN)) {
        /* ACPI is not enabled;
             this interrupt cannot be for us */
        return_VALUE (INTERRUPT_NOT_HANDLED);
    }

    /*
     * Fixed Acpi_events:
     * -------------
     * Check for and dispatch any Fixed Acpi_events that have
     *  occurred
     */
    interrupt_handled |= acpi_ev_fixed_event_detect ();

    /*
     * GPEs:
     * -----
     * Check for and dispatch any GPEs that have occurred
     */
    interrupt_handled |= acpi_ev_gpe_detect ();

    return_VALUE (interrupt_handled);
}
```

〔図5〕ACPIイベントを発生させ，KDBで動作を追跡する

```
[0]kdb> bp acpi_ev_sci_handler
Instruction(i) BP #0 at 0xc01be290 (acpi_ev_sci_handler) is enabled globally adjust 1

[0]kdb> bp acpi_ev_fixed_event_detect
Instruction(i) BP #1 at 0xc01bbfa0 (acpi_ev_fixed_event_detect) is enabled globally adjust 1

[0]kdb> bp acpi_ev_gpe_detect
Instruction(i) BP #2 at 0xc01bc750 (acpi_ev_gpe_detect) is enabled globally adjust 1

[0]kdb> go
```

〔図6〕ブレークポイント0のSCI割り込みサービスルーチンでブレークしたようす

```
Instruction(i) breakpoint #0 at 0xc01be290
 (adjusted)
0xc01be290 acpi_ev_sci_handler:         int3

Entering kdb (current=0xc046e000, pid 0) on processor 0 due to Breakpoint @ 0xc01be290

[0]kdb>
```

〔リスト 2〕os.c

```
acpi_status
acpi_os_read_port(
    ACPI_IO_ADDRESS port,
    void    *value,
    u32     width)
{
    u32 dummy;

    if (!value)
        value = &dummy;

    switch (width)
    {
    case 8:
        *(u8*)  value = inb(port);
        break;
    case 16:
        *(u16*) value = inw(port);
        break;
    case 32:
        *(u32*) value = inl(port);
        break;
    default:
        BUG();
    }

    return AE_OK;
}

acpi_status
acpi_os_write_port(
    ACPI_IO_ADDRESS port,
    NATIVE_UINT value,
    u32     width)
{
    switch (width)
    {
    case 8:
        outb(value, port);
        break;
    case 16:
        outw(value, port);
        break;
    case 32:
        outl(value, port);
        break;
    default:
        BUG();
    }

    return AE_OK;
}
```

〔リスト 3〕デバッグログ

```
[0]kdb> bt  // コールバックトレース表示
EBP        EIP         Function(args)
0xc046fe08 0xc01af448  acpi_os_read_port+0x58 (0xf100,
         0xc046fe28, 0x10,,, // 読み出し I/O ポートアドレス,
                             // 読み出し値の格納先, 読み出す長さ
0xc046fe38 0xc01c1629  acpi_hw_low_level_read+0xe9 (,,,,,
0xc046fe6c 0xc01c1007  acpi_hw_register_read+0xe7 (,,,,,
0xc046fea8 0xc01bbfd5  acpi_ev_fixed_event_detect+0x35 (,,,,,
0xc046fed4 0xc01be315  acpi_ev_sci_handler+0x85 (,,,,,
0xc046ff00 0xc010973b  handler_IRQ_event+0x6b (,,,,,
  ....

[0]kdb> rd  // レジスタの状態一覧
eax = 0xc046fe00  ebx = 0x0000f100  ecx = 0xc046fe28
  edx = 0x0000f100
esi = 0xc7f794bc  edi = 0x0000f100  esp = 0xc046fe00
  eip = 0xc01af448
ebp = 0xc046fe08
  ....

[0]kdb> id 0xc01af448 // これから実行する命令のディスアセンブル

0xc01af448 acpi_os_read_port+0x58:    in     (%dx),%ax
 // I/O ポート 0xf100 から 1 word 分読み出し,
0xc01af44a acpi_os_read_port+0x5a:    mov    %ax,(%ecx)
 // ecx がポイントするアドレスに格納.
0xc01af44d acpi_os_read_port+0x5d:    jmp    0xc01af421
                                          acpi_os_read_port+0x31
  ....

[0]kdb> ssb  // 次のブランチまで実行

0xc01af44d acpi_os_read_port+0x5d:    jmp    0xc01af421
                                          acpi_os_read_port+0x31
[0]kdb> rd
eax = 0xc0461111  ebx = 0x0000f100  ecx = 0xc046fe28
  edx = 0x0000f100
esi = 0xc7f794bc  edi = 0x0000f100  esp = 0xc046fe00
  eip = 0xc01af44d
ebp = 0xc046fe08
  ....

// ACPI Debugger 機能の使用例
[0]kdb> acpi  // KDB から ACPI Debugger に入る
- Tables  // ACPI CA サブシステムがメモリ内にコピーした
          // ACPI テーブル群の先頭アドレスとテーブル長を列挙

DSDT at c88020cc length 3E5B
FACP at c7f79428 length F4
  ....

- q  // KDB に戻る
```

過程を割り込みサービスルーチン内で行うわけにいきません.

イベント種別が汎用イベントだと，そのハンドリングは複数のステージにまたがるので，本質的な部分は割り込みのボトムハーフ処理（Delayed Procedure Call : DPC）に回します.

OSは割り込みのトップハーフで，汎用イベントが発生しているビット位置を特定し，そのビットに呼応するGPEハンドラの実行をキューイングします．そしてボトムハーフの処理が走れるようになったら，AMLインタプリタを用いてキューイングされたGPEハンドラを実行していくわけです.

なお，汎用イベントを収集するレジスタの各ビットにどんなイベントを結びつけるかは，マシンの製造元がそれこそ「汎用」的に自由に決定することができます.

リスト 4のように，ACPI OSに通知したい汎用イベントがあるならば，まずそれをルーティングする先のビットと，レベルで入るのかエッジで入るのかを定義します．そして，その汎用イベントで実際に伝えたい内容は何か，またどんなアクションをOSにしてもらいたいのかをGPEハンドラ内に記述します.

リスト 4では，どれも単純にイベントの発生源であるデバイスと，イベントの中身をASLのNotify命令でOS内の関係するエンティティに通知しているだけです.

しかし，現実のACPI BIOSでは，もっと複雑に，いろいろなレジスタを参照して条件判断しています．あるいは，複数のイベントをいったん組み込みコントローラに集約して，OSへは組み込みコントローラからの汎用イベントとしてまず通知し，その後，AMLインタプリタからGPEハンドラコードを介してクエリーをかけてイベント種別とそのハンドリングを特定するようなこともよくあります（**図 7**).

このあたりはぜひ，前編で紹介した米国Intel製のAMLディスアセンブラ（`http://developer.intel.com/technology/iapc/acpi/downloads.htm`）で，お使いの

〔リスト4〕ACPI OSに通知したい汎用イベント

```
Scope (¥_GPE)  // GPE ハンドラ群の非常にシンプルな実装例
{
    Method (_L00, 0, NotSerialized)
    // GPE pin 0x0 にレベルで入った汎用イベント用のハンドラ
    {
     Notify (¥_TZ.THR0, 0x80)
     // 放熱管理のための Thermal Zone における温度変化
    }

    Method (_L01, 0, NotSerialized)
    // GPE pin 0x1 にレベルで入った汎用イベント用のハンドラ
    {
     Notify (¥_PR.CPU0, 0x80)
     // Intel SpeedStep などの CPU パフォーマンス制御でサポート
     // される最高ステートの切り替え
    }

    Method (_L03, 0, NotSerialized)
    // GPE pin 0x3 にレベルで入った汎用イベント用のハンドラ
    {
        Notify (¥_SB.PCI0.USB0, 0x02)
        // USB ホストコントローラ USB0 配下のデバイスが発生
        // させたウェイクイベント
    }

    Method (_L0B, 0, NotSerialized)
    // GPE pin 0xB にレベルで入った汎用イベント用のハンドラ
    {
        Notify (¥_SB.SLPB, 0x02)
       // スリープボタン押し下げによって発生したウェイクイベント
    }
}
```

〔図7〕汎用イベント(GPE)のハンドリング

PCのASLコードを逆ダンプし，¥_GPEのスコープにあるGPEハンドラを確認してみてください．

Windows版では，コマンドラインからiasl -dとすれば，レジストリ

HKEY_LOCAL_MACHINE¥HARDWARE¥ACPI¥DSDT

の下にコピーされたDSDTの内容がディスアセンブルされました．

またLinuxであれば，/proc/acpi/の下にDSDTのコピーが公開されていますから，

```
# iasl -d /proc/acpi/dsdt
```

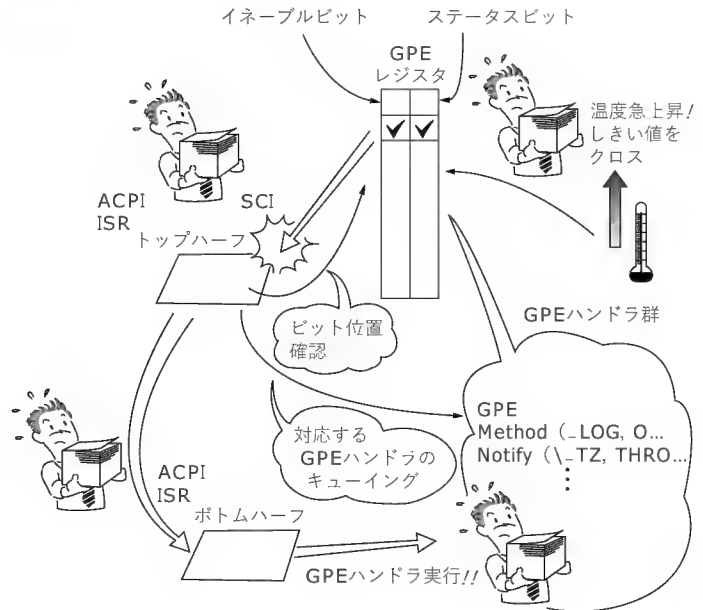
とすればOKです．

▶ユーザーランドへの通知

ここまでが，カーネルモードでACPIイベントが処理されていくようすです．もちろん，いくつかのACPIイベントは，その発生がユーザーランドに伝達され，適切なアクションが起動されることに意義があります．

電源ボタン押し下げでシャットダウンさせるケースなどは典型的な例ですが，ほかにもパターンは多数あります．

たとえば，電源に接続した状態からバッテリ駆動に切り替えたときに，Intel SpeedStepなどのプロセッサ省電力テクノロジでサポートされる最高動作ステートをダウングレードさせたり，放熱のためにファンの回転を許可したり，またディスプレイの輝度を落とすといったことができます．

あるいはバッテリの残量低下にあわせて，システムを休止状態(Suspend To Disk : S4)へ移行させるといったこともACPIイベントと，そうしたパワーマネージメントアクションをリンクさせることで可能になります．

そのためにも，カーネルモードでACPIイベントを検知したら，それをユーザーランドに通知してあげなければなりません．

Linuxでは，/proc/acpi/eventに，発生したACPIイベントを書き込むことでこれを実現しています．

ただし現状の課題は，このエントリを自分で監視するか，あるいはそれをすでに取り入れているデーモンであるacpidを利用して，その制御スクリプトに手を入れることぐらいしか，便利でかつ拡張性の高い電力管理の方法がありません．

C言語ベースでlibpowerというライブラリも存在しますが，メンテナンスの状態はあまり芳しくないようです．

ACPIイベントのハンドル――Windows編

ここまでは，オープンソースであるLinuxを題材にして，ACPIのイベントハンドリングのようすをカーネルデバッガを併用し解説してきましたが，Windows XPでは，このACPIイベントハンドリングをどのようにしているのでしょうか．Windows XPでの例を見てみましょう．

● Windows XPの観察

▶必要なソフトウェアの入手

Windowsファミリのカーネルデバッグには，製造元であるMicrosoftが公開している最新のカーネルデバッガとシンボルが必須です．

まずデバッガを次のサイトからダウンロードしてインストールしましょう．

● Microsoft Debugging Tools

http://www.microsoft.com/whdc/ddk/debugging/default.mspx

シンボルは，ターゲットOSのバージョンにあわせて，次のサイトからダウンロードするか，あるいはWebベースで公開されているものを利用することもできます．

● How to Get Symbols

http://www.microsoft.com/whdc/ddk/debugging/symbols.mspx

デバッガのインストールが完了したら，カーネルデバッグの設定をターゲットとホストの両方で行います．

Windows XP以降だったら，シリアルのクロスケーブルだけでなくIEEE1394接続でデバッグすることもできます．

また，カーネルデバッグ用のデバッガには，GUIベースのWinDbgとキャラクタベースのKDがありますが，現在はどちらも共通のエンジンを使用しているため，操作性以外の性能差はありません．今回は説明が容易なため，KDでシリアル接続する前提で進めていきます．

カーネルデバッグの設定自体は，ダウンロードしたデバッガパッケージに含まれるヘルプファイルdebugger.chmを参照してください．また，WinDbgで操作したい場合や，COMポートがないなどの理由でIEEE1394接続での方法を試す必要がある場合も，同様にヘルプを参照してください．従来とは異なり，ヘルプもたいへん充実した内容となっているので，開発者であれば設定にとまどうことはないと思われます．

▶ターゲットマシンの起動

それでは，準備が整ったところで，ターゲットマシンをデバッグ有効モードで起動します．

このとき，ホスト側は，以下のようにシンボルパス，デバッグ用のポートおよび通信速度が環境変数としてセットされた状態で待ち受けています．

```
> set
_NT_SYMBOL_PATH=SRV*C:¥Websymbols*http://
msdl.microsoft.com/download/symbols // ①
> set _NT_DEBUG_PORT=COM1
> set _NT_DEBUG_BAUD_RATE=115200
> kd -d

Microsoft(R)Windows Debugger ……
Copyright(c)Microsoft Corporation ……

Opened¥¥.¥COM1
Waiting to reconnect...
```

この例では，①のように米国MicrosoftがWebベースで公開しているシンボルサーバをシンボルパスとして指すようにしてみました．こうすると，バージョンの合致したシンボルがデバッガ上でロードされ，かつ指定したローカルフォルダ（上ではC:¥Websymbols¥）にキャッシュされるので，ネットワークの速度に問題がなければ十分使用に耐えます．

なお，以下で紹介するコマンドは一般に公開されているデバッガとシンボルで実行可能なものばかりです．またほとんどすべてについて，ヘルプファイルに記載があります．詳細について不明なときは，ヘルプファイルを確認してください．

▶基本コマンドの使い方

まず，SCIを発行するACPIレジスタに割り当てられた割り込みベクタと，割り込みサービスルーチンを特定しなくてはいけません．

これには!idtコマンドか!arbiterコマンドを用います．ホスト側でCtrl+Cキーを押し下げブレークインし，コマンドを実行してください．

実行例を**リスト5**に示します．すると，ACPIはIRQ 9に割り振られており，SCIを受け付ける割り込みサービスルーチンは，ACPIドライバのACPIInterruptServiceRoutine()という関数であることがわかります．この関数にブレークポイントを設定しておきます．

```
kd> bp ACPI!ACPIInterruptServiceRoutine
kd> g

Breakpoint 0 hit
ACPI!ACPIInterruptServiceRoutine:
fc43de3e 55              push  ebp
```

そして，このブレークポイントに引っ掛かったら，リターンアドレスまでwtコマンドで走らせてみましょう．

```
kd> k1
ChildEBP RetAddr
80543e4c 804f20ba
          ACPI!ACPIInterrupt ServiceRoutine
kd> wt 804f20ba
```

実行結果は**リスト5**にあるとおりです．このwtコマンドによる実行追跡結果を見ると，やはりLinux同様，イベントが固定イベントか汎用イベントか，また固定イベントだとして，Power Managementイベントブロックのどこのビットがセットされているのか，といった点をチェックしているであろうことが容易に推察されます．

▶その他のコマンド

イベント種別を判定するには，デバッガで先回りしてFADTからたどり，Power Managementイベントブロックの状態を確認しておくこともできます．

そうすれば，OS側がどのような判定を下すかあらかじめ予想がつくことになり，便利です．

Windowsが内部に保持するFADTの内容は，ずばり!fadtコマンドで得られます．

```
kd> !fadt
 ……
```

〔リスト5〕コマンドの実行例

```
C:¥Debuggers> set _NT_SYMBOL_PATH=SRV*C:¥Websymbols*http://msdl.microsoft.com/download/symbols
C:¥Debuggers> set _NT_DEBUG_PORT=COM1
C:¥Debuggers> set _NT_DEBUG_BAUD_RATE=115200
C:¥Debuggers> kd -b -d

Microsoft (R) Windows Debugger  Version 6.2.0007.4
Copyright (c) Microsoft Corporation. All rights reserved.

Opened ¥¥.¥COM1
Waiting to reconnect...
Connected to Windows XP 2600 x86 compatible target, ptr64 FALSE
Kernel Debugger connection established.  (Initial Breakpoint requested)
Symbol search path is: SRV*C:¥Websymbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows XP Kernel Version 2600 UP Free x86 compatible
Built by: 2600.xpclient.010817-1148
Kernel base = 0x804d8000 PsLoadedModuleList = 0x8054db28
System Uptime: not available
nt!DebugService2+e:
8050e563 cc               int     3
kd> g  // 起動時にいったんブレークするよう設定していた( -b )ためブレーク．スリープからのウェイク時などに便利．
       // 必要がなければ何もせずそのまま go

// Ctrl+C でブレークイン

Break instruction exception - code 80000003 (first chance)
*******************************************************************************
*                                                                             *
*   You are seeing this message because you pressed either                    *
*       CTRL+C (if you run kd.exe) or,                                        *
*       CTRL+BREAK (if you run WinDBG),                                       *
*   on your debugger machine's keyboard.                                      *
*                                                                             *
*                   THIS IS NOT A BUG OR A SYSTEM CRASH                       *
*                                                                             *
* If you did not intend to break into the debugger, press the "g" key, then   *
* press the "Enter" key now.  This message might immediately reappear.  If it *
* does, press "g" and "Enter" again.                                          *
*                                                                             *
*******************************************************************************
nt!RtlpBreakWithStatusInstruction:
80515064 cc               int     3
kd> !idt  // ACPI にアサインされた割り込みベクタと割り込みサービスルーチンの確認

Dumping IDT:

30:     806c26e4 hal!HalpClockInterrupt
31:     ffb2d9dc i8042prt!I8042KeyboardInterruptService (KINTERRUPT ffb2d9a0)
38:     806bd160 hal!HalpProfileInterrupt
39:     80e977dc ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 80e977a0)
3a:     ff9ff414 NDIS!ndisMIsr (KINTERRUPT ff9ff3d8)
                 portcls!CInterruptSync::Release+0x10 (KINTERRUPT ff9fba90)
3b:     80e933d4 ohci1394!OhciIsr (KINTERRUPT 80e93398)
                 pcmcia!PcmciaInterrupt (KINTERRUPT 80e0dd98)
                 pcmcia!PcmciaInterrupt (KINTERRUPT 80e0da68)
                 VIDEOPRT!pVideoPortInterrupt (KINTERRUPT ff9ff648)
                 USBPORT!USBPORT_InterruptService (KINTERRUPT ffb2c298)
                 USBPORT!USBPORT_InterruptService (KINTERRUPT ffbbac10)
3c:     80d798e4 i8042prt!I8042MouseInterruptService (KINTERRUPT 80d798a8)
3e:     80e0c9d4 atapi!IdePortInterrupt (KINTERRUPT 80e0c998)
3f:     80e39044 atapi!IdePortInterrupt (KINTERRUPT 80e39008)

kd> !arbiter 4  // 割り込みリソースの確認

DEVNODE 80e753d0 (HTREE¥ROOT¥0)
  Interrupt Arbiter "RootIRQ" at 8054b960
    Allocated ranges:
     . . . .
      0000000000000039 - 0000000000000039 S      80e70aa8  (ACPI)
    Possible allocation:
      < none >

    DEVNODE 80e703e8 (ACPI_HAL¥PNP0C08¥0)
      Interrupt Arbiter "" at fc473780
        Allocated ranges:
        . . . .
          0000000000000009 - 0000000000000009 S      80e70aa8  (ACPI)


kd> bp ACPI!ACPIInterruptServiceRoutine  // ACPI からの割り込み (=SCI) 用のサービスルーチン
kd> g
```

〔リスト5〕コマンドの実行例（つづき）

```
Breakpoint 0 hit
ACPI!ACPIInterruptServiceRoutine:
fc43de3e 55              push    ebp
kd> kv1
ChildEBP RetAddr  Args to Child
80543e4c 804f20ba 80e977a0 80e6fea0 8054c302
ACPI!ACPIInterruptServiceRoutine (FPO: [Non-Fpo])
kd> wt 804f20ba  // リターンするまでの呼び出し関係をトレースし，統計的に解析
    5     0 [  0] ACPI!ACPIInterruptServiceRoutine
    1     0 [  1]   ACPI!ACPIIoReadPm1Status
    8     0 [  2]     ACPI!READ_PM1_STATUS
   10     0 [  3]       ACPI!DefReadAcpiRegister
    4     0 [  4]         hal!READ_PORT_USHORT
   13     4 [  3]       ACPI!DefReadAcpiRegister
   15    17 [  2]     ACPI!READ_PM1_STATUS
    7    32 [  1]   ACPI!ACPIIoReadPm1Status
    7    39 [  0] ACPI!ACPIInterruptServiceRoutine
    7     0 [  1]   ACPI!ACPIGpeIsEvent
    5     0 [  2]     ACPI!ACPIReadGpeStatusRegister
   15     0 [  3]       ACPI!DefReadAcpiRegister
    4     0 [  4]         hal!READ_PORT_UCHAR
   19     4 [  3]       ACPI!DefReadAcpiRegister
    7    23 [  2]     ACPI!ACPIReadGpeStatusRegister
   17    30 [  1]   ACPI!ACPIGpeIsEvent
    5     0 [  2]     ACPI!ACPIReadGpeStatusRegister
   15     0 [  3]       ACPI!DefReadAcpiRegister
    4     0 [  4]         hal!READ_PORT_UCHAR
   19     4 [  3]       ACPI!DefReadAcpiRegister
    7    23 [  2]     ACPI!ACPIReadGpeStatusRegister
   27    60 [  1]   ACPI!ACPIGpeIsEvent
    5     0 [  2]     ACPI!ACPIReadGpeStatusRegister
   18     0 [  3]       ACPI!DefReadAcpiRegister
    4     0 [  4]         hal!READ_PORT_UCHAR
   22     4 [  3]       ACPI!DefReadAcpiRegister
    7    26 [  2]     ACPI!ACPIReadGpeStatusRegister
   37    93 [  1]   ACPI!ACPIGpeIsEvent
    5     0 [  2]     ACPI!ACPIReadGpeStatusRegister
   18     0 [  3]       ACPI!DefReadAcpiRegister
    4     0 [  4]         hal!READ_PORT_UCHAR
   22     4 [  3]       ACPI!DefReadAcpiRegister
    7    26 [  2]     ACPI!ACPIReadGpeStatusRegister
   48   126 [  1]   ACPI!ACPIGpeIsEvent
   18   213 [  0] ACPI!ACPIInterruptServiceRoutine
    9     0 [  1]   ACPI!CLEAR_PM1_STATUS_BITS
   11     0 [  2]     ACPI!DefWriteAcpiRegister
    4     0 [  3]       hal!WRITE_PORT_USHORT
   14     4 [  2]     ACPI!DefWriteAcpiRegister
   14    18 [  1]   ACPI!CLEAR_PM1_STATUS_BITS
   22   245 [  0] ACPI!ACPIInterruptServiceRoutine
   18     0 [  1]   hal!HalAcpiTimerCarry
   35   263 [  0] ACPI!ACPIInterruptServiceRoutine
298 instructions were executed in 13 events (0 from other threads)

Function Name                     nvocations MinInst MaxInst AvgInst
ACPI!ACPIGpeIsEvent                        1      48      48      48
ACPI!ACPIInterruptServiceRoutine           1      35      35      35
ACPI!ACPIIoReadPm1Status                   1       7       7       7
ACPI!ACPIReadGpeStatusRegister             4       7       7       7
ACPI!CLEAR_PM1_STATUS_BITS                 1      14      14      14
ACPI!DefReadAcpiRegister                   5      13      22      19
ACPI!DefWriteAcpiRegister                  1      14      14      14
ACPI!READ_PM1_STATUS                       1      15      15      15
hal!HalAcpiTimerCarry                      1      18      18      18
hal!READ_PORT_UCHAR                        4       4       4       4
hal!READ_PORT_USHORT                       1       4       4       4
hal!WRITE_PORT_USHORT                      1       4       4       4

nt!KiInterruptDispatch+2a:
804f20ba fa              cli
kd>

kd> !fadt  // FADT のコピーを表示
FADT -- 806c3c20HEADER - ffffffff806c3c20
  Signature:               FACP
  Length:                  0x00000074
  Revision:                0x01
  Checksum:                0x6e
  OEMID:                   CQPUB
  OEMTableID:              SAMPLE
  OEMRevision:             0x00000001
  CreatorID:               CQ
  CreatorRev:              0x00000000
FADT - BODY - ffffffff806c3c44
  FACS:                       0x07fe8000
  DSDT:                       0x07fe00cc
  Int Model:                  Dual PIC
  SCI Vector:                 0x009
  SMI Port:                   0x000000b2
  ACPI On Value:              0x055
  ACPI Off Value:             0x0aa
  SMI CMD For S4 State:       0x077
  PM1A Event Block:           0x0000f100
  PM1B Event Block:           0x00000000
  PM1 Event Length:           0x004
  PM1A Control Block:         0x0000f104
  PM1B Control Block:         0x00000000
  PM1 Control Length:         0x002
  PM2 Control Block:          0x00000000
  PM2 Control Length:         0x000
  PM Timer Block:             0x0000f108
  PM Timer Length:            0x004
  GP0 Block:                  0x0000f128
  GP0 Length:                 0x004
  GP1 Block:                  0x0000f12c
  GP1 Length:                 0x00000004
  GP1 Base:                   0x00000010
  C2 Latency:                 0x0ffff
  C3 Latency:                 0x003e9
  Memory Flush Size:          0x00000
  Memory Flush Stride:        0x00000
  Duty Cycle Index:           0x000
  Duty Cycle Index Width:     0x000
  Day Alarm Index:            0x00d
  Month Alarm Index:          0x000
  Century byte (CMOS):        0x032
  Boot Architecture:          0x0000
  Flags:                      0x000000ad
    Write Back Invalidate is supported
    System supports C1 Power state on all processors
    System supports C2 in MP and UP configurations
    Sleep Button is treated as a generic feature
    RTC Wake can work from an S4 state

kd> id f100  // デバッガから I/O ポートを読み書きできる
0000f100: 01210011
kd> !acpiinf  // ACPI 関連の主要な情報一覧
ACPIInformation (80e6fce0)
  RSDT                        - fca8e000
  FADT                        - fca90054
  FACS                        - fca91000
  DSDT                        - fc9400cc
  GlobalLock                  - fca91010
  GlobalLockQueue             - F - 80e6fcf8 B - 80e6fcf8
  GlobalLockQueueLock         - 00000000
  GlobalLockOwnerContext      - 00000000
  GlobalLockOwnerDepth        - 00000000
  ACPIOnly                    - FALSE
  PM1a_BLK                    - 0000f100 (0011) (0121)
     1 - TMR_STS  10 - BM_STS
     1 - TMR_EN  20 - GBL_EN  100 - PWRBTN_EN
  PM1b_BLK                    - 00000000 (N/A)
  PM1a_CTRL_BLK               - 0000f104 (1c01)
     1 - SCI_EN
  PM1b_CTRL_BLK               - 00000000 (N/A)
  PM2_CTRL_BLK                - 00000000 (N/A)
  PM_TMR                      - 0000f108 (00bb6cb1)
  GP0_BLK                     - 0000f128 (00) (01)
  GP0_ENABLE                  - 0000f12a (19) (00)
  GP0_LEN                     - 00000004
  GP0_SIZE                    - 00000002
  GP1_BLK                     - 0000f12c (7f) (38)
  GP1_ENABLE                  - 0000f12e (00) (21)
  GP1_LEN                     - 4
  GP1_SIZE                    - 2
  GP1_BASE_INDEX              - 10
  GPE_SIZE                    - 4
  PM1_EN_BITS                 - 0121
     1 - TMR_EN  20 - GBL_EN  100 - PWRBTN_EN
  PM1_WAKE_MASK               - 0000
  C2_LATENCY                  - 0
  C3_LATENCY                  - 0
  ACPI_FLAGS                  - 0
  ACPI_CAPABILITIES           - 0
```

```
PM1A Event Block:        0x0000f100
  ……
```

そして，Power Managementイベントブロックの状態を見るには，!fadtコマンドの出力結果のうち，PM1A Event Blockエントリが示すI/Oポートの値0x0000f100をidコマンドで読み出せばよいわけです．

```
kd> id f100
0000f100: 01210011
```

あるいは，!acpiinfコマンドを用いると，同じデータですが，ビットの意味をデコードして出力してくれます．

```
kd> !acpiinf
  ……
 PM1a_BLK            -0000f100(0011) (0121)
  1-TMR_STS 10-BM_STS
  1-TMR_EN 20-GBL_EN 100-PWRBTN_EN
  ……
```

上の状態だと，固定イベントとしての電源ボタン押し下げイベントは，入ってくればOSに通知するようイネーブルになっていますが，ステータスビットはセットされておらず，現在イベントは上がっていないことになります．

なお，今回使用した最新のWindowsカーネルデバッガには，上に例を示したようなACPIに関わる便利なコマンドが内蔵されているほか，非常に高機能な"AMLI Debugger"も組み込まれています．

これにより，一般の"フリービルド"と呼ばれるターゲットを用いた場合でも，ACPIネームスペース内のオブジェクトの検索やACPIコントロールメソッドの実行トレースができます．また，ACPIドライバ(acpi.sys)を"チェックドビルド"と呼ばれるデバッグ用機能を残したものに差し替えると，ACPIコントロールメソッドの任意のステートメントでブレークさせてステップ実行させたり，引き数やローカル変数の値を操作するといった，一般のデバッガでできるようなことがAMLインタプリタに対して行えるようになります．

Windows製品のチェックドビルドは，米国MicrosoftのMSDN有料会員になることで入手が可能です．シンボルはフリービルド，チェックドビルドともに無償で提供されています．

ACPI関係のデバッガコマンドやAMLI Debuggerの詳細については，デバッガパッケージに含まれるヘルプファイル，debugger.chmをご確認ください．

なお，Linuxのほうでも，今回取り上げたバージョン2.4.20あたりまではKDBと連携して動作可能なACPI Debugger機能が存在しますが，Windowsの同等機能と比較するとかなりシンプルなものです．

また，このLinux版ACPI Debuggerは，メンテナンスの都合もあり，現在のメインのツリーからはドロップされています．

▶まとめ

ごく簡単でしたが，WindowsもおおむねLinuxと同じようにACPIイベントをハンドルしていそうなことが理解できたのではないかと思います．

なお，Windowsの場合は，カーネル内に存在するパワーマネージメントマネージャが，発生したACPIイベントを拾って，カーネル内部で取り決めたアクションや，ユーザー(プログラムを含む)の指定したアクションを起動するしくみがとても柔軟で，SDK/DDKに公開されているドキュメントを参照してコーディングすれば，イベントとアクションの非常に多彩な組み合わせを実現できます．こうした点については，機会があれば，稿をあらためてプログラミングTipsなどを紹介したいと思います．

また，Linux側でACPIが真にパワーマネージメントの土台となるには，カーネル側の充実もさることながら，ユーザーランドでのケアも重要だと思います．こちらもまた，機会があれば，いくつかの腹案を実行に移して，読者の皆様に成果を還元できればと考えています．

おわりに

今回は，当初の予定を大きく変更して，ACPIイベントハンドリングの追跡に終始してしまいましたが，いかがだったでしょうか．

できればシステムスリープステートへの遷移や，プロセッサドライバといったトピックにも触れたかったのですが，残念ながら今回は見送りたいと思います．

あだち・けんいち

● 悪夢からの開放

先日，思い切って蔵書を捨ててしまった．ほとんど全部．理由の一つは，夜寝ていて，高くそびえる三つの本棚が精神衛生上よろしくなかったからである．狭い部屋の中で囲まれた本に押しつぶされる悪夢を，そのうち見る気がしていたからである．三つの本棚の中身がほとんど空になってしまったので，ついでに本棚も捨ててしまった．その代わり，鉄パイプを組み合わせたようなインテリアラックを買い，残った本や小物を置いてみた．なかなかスマートでいい．リフレッシュされた気分だ．

本を捨てた理由はほかにもある．技術関係の本はいつの間にか山のように買っていたのだが，惰性で毎月買っている雑誌も多かった．買ったあと一度も開いていない本も多かった．そこで，この際一気に整理することにしたのだ．

極端な例では，あるベストセラーマイコン時代の末期に出た，ハードカバーの赤い本もあった．当時で9,800円もした記憶がある．その本は，マイコンのソースコードが，たんたんと記されているという，ただそれだけの本だ．一部の読者には記憶がある方もおられるかもしれない．

この本は買ったあと，たぶん一度も開いた記憶がない．そして，きっとこれからも開くことはないだろう．こんな感じだったから，長い間本を買い続けてきて，冷静に本棚を眺めてみるに，あまりにも非効率に本を買ってきた自分が嘆かわしかった．

本を思い切って捨ててから，ここのところずっと本は買っていない．いや，欲しいと思っても買わないことにしている．ほとんどの場合，買わなくても事足りることに気がついたからだ．どうしても情報が欲しければ，インターネットで多くの情報がまとめられているから問題はおきない．

● ディジタル万引き

出版不況といわれて久しい．とくに雑誌はかなり売れ行きが落ちているそうだ．そんな中で，最近問題になっているのが「ディジタル万引き」というものだそうだ．カメラ付き携帯の普及で，本屋へ行き必要なページのみ携帯で撮影してくる客が増えているということだ．「撮影するだけで，本を買わない人が増えたから売り上げが落ちて困る．しかし，本を見ながらメールを見ている人もいるので簡単には注意できない」と本屋の主人は言う．本を見ながらメールしている人がいるかどうかは別問題として，本屋さんはたしかに困っているようだ．業界専門誌『新文化』によれば，全国の書店の廃業数は2002年度で6年連続1,000店を越えているらしい．

日本雑誌協会では，「ディジタル万引き」に注意を促すポスターを全国の書店に3万枚も張り出し，雑誌記事を撮影しないように呼びかけているらしい．日本雑誌協会の話によれば，「今日食べに行くお店や，映画など日常に密着した情報がカメラ付き携帯電話で撮影されてしまう．すべてをカメラ付き携帯電話のせいにするつもりはないが，ディジタル万引きについて，小売書店から多数の苦情が上がっており，メガピクセル携帯電話の登場でこうした行為が加速する」と警戒感を示している．

また，今回のキャンペーンには，電気通信事業者協会も協賛しているようだ．ある種責任の一端を感じてのことか，それとも責任逃れに先手を打ったとでもいうのか．

● 著作権と肖像権の侵害

しかし，出版業界が静かなのはおかしい．出版業界も大きな被害を受けているだろうに……．雑誌の撮影は，著作権，肖像権の侵害の可能性があるというのが今回の「ディジタル万引」を追及する根拠のようだ．しかし，著作権や肖像権の問題では，出版社自身が過去に何度も訴えられている．小説などの引用で著作権違反で告訴されるのは珍しいことでもない．また写真週刊誌なども，肖像権の侵害などで告訴されたことは，まだ記憶に新しい．

このように出版社は，事業として大量に印刷し，直接利益を得るだけでなく，社会的影響力も大きい．そのため，著作権や肖像権に最大限配慮すべきだろう．しかし，自分で使用するために，個人が携帯で雑誌を撮影するのは違法なのだろうか？

専門家ではないから詳しいことはわからないが，著作権法30条によれば，私的使用のための複製は違反ではない．しかし，政令に定めるディジタル機器による複製を除くという例外事項がある．これは音楽専用CDレコーダやDAT，MD，DCCなどが指定されている．CD-R/RWやデジカメ，携帯は政令で指定されたディジタル機器ではないので，これらを使用して撮影しても違法ではない．

もっとも，音楽のように，曲全体がまったく同じ品質のままコピーされ，無料で流通してしまうような問題とは一線を画す

るはずだ．一緒に議論されても困る．カメラ付き携帯やデジカメでは，1回1ページしか取れないし，流通させることができるほど高画質で撮るのは難しいからだ．

また，肖像権は，憲法13条で，幸福追求に対する国民の権利が記述されていることに起因する人格権の一つとして規定されているにすぎず，ほかに具体的な法令はない．したがって，過去の判例のみで確立されているが，すべてパブリシティ（印刷物）に対する肖像権者の権利という形であり，一般個人が携帯で撮影し自分で使用する場合に肖像権が問題になるかどうかはもちろん判例がなく，推し量る手段もない．

紀藤正樹弁護士がTVで次のようなことをいっていた．「携帯で本を撮影する行為は，記憶の延長にすぎないので，違法ではない」．彼は，TVでも一般の消費者問題などをわかりやすく解説してくれるが，じつはネット問題やマルチメディア関連の専門家でもある．

万引きは犯罪だ．しかし，現時点では，ディジタル万引きは犯罪ではない．とすると，この言葉の使い方は教育上ちょっとよろしくないのではないだろうか．正しくは，「ディジタル立ち読み」ではないのか．雑誌の内容を立ち読みした記憶を，カメラ付き携帯やデジカメで持ち帰るだけなのだから．

立ち読みがマナーとして良くないということは，当然広く知れ渡っている常識だ．だからディジタル立ち読みは遠慮しなさいというのはわかる．それでは，なぜ「ディジタル立ち読みは止めましょう」といわないのか．

「立ち読み」を電子辞書でネットの和英辞典で引いてみた．「ブラウジング」というのだそうだ．すると，ブラウザというのは，立ち読みするソフトだったのか……．だからインターネットは無料だったのだ．

● 買う価値のある雑誌

100ページ以上ある雑誌のうち，数ページだけ欲しいのでその雑誌を買ったとしても，無駄なお金を使ったという感覚しか残らないだろう．これを続けていると，その雑誌はそのうち買わなくなるのではないか．

ある女性は，花火大会の日程が載っているところをカメラ付き携帯で撮影したという．また，ある男性は，ラーメンショップで自宅に近いところがあったので撮影したという．どうも，何ページも無差別に撮影しているわけでなく，必要な1，2ページしか撮影していないようだ（誰でもそんなにいっぱい撮影すると気が惹ける）．

しかし，この程度の情報は，インターネットで公開されていることが多い．生意気かもしれないが，カメラ付き携帯で取られただけで売り上げが落ちるような雑誌は，作っている側にも責任の一端があるのではないかと思うが，どうだろうか？

本当は，「携帯でどんどん撮影してかまいません」としたほうが，人が集まって本の売り上げが増える気もする．なにせ日本の携帯電話の普及数は8000万台で，ほとんどがカメラ付き携帯電話なのだから．

本棚を捨ててしまった筆者の部屋は，とても広くなった．これからは，本も雑誌も本屋でじっくり見て，よく考えてから最小限のものを買うことにする．これで，もう本が落ちてくる夢など見ることはないだろう．毎晩安心して眠りにつける．

あさひ・しょうすけ　テクニカルライター
イラスト　森　祐子

Engineering Life in

放浪の旅を経てエンジニアに……

今回のゲストのプロフィール
下村朋子(しもむらゆうこ):日本での営業職に疑問を感じ，自分のやりたいことを探す旅に出て，オーストラリア，カナダ，アメリカと移り住み，現在はシリコンバレーに落ち着いている．南カリフォルニア，サンディエゴ市のサンディエゴ州立大学で情報システムの学士を取得．その後，インターネットバブルの波にのってシリコンバレーへ乗り込み，スタートアップでEコマースのサーバサイドのバックエンド構築に従事する．趣味は旅行．アフリカ大陸最高峰キリマンジャロに登頂．その他，コスタリカ，中国，メキシコ，ヨーロッパなど．yuko-sv@sbcglobal.net

☆ 放浪の旅へ……プログラミングと出会う

トニー さて，アメリカで大学を出られた日本人エンジニアが久々に登場したので，まずはアメリカに留学されたきっかけからお話いただけますか？

下村 少し長くなりますが，高校時代の話から始めます．

私は入った高校が合わず，中退して広告代理店の営業として就職しました．仕事は楽しかったのですが，それでもやはり高校を卒業したいと思い，夜間高校に入りました．夜学の担任の先生が大の旅行好きで，フィリピンの山奥に一人で蝶を取りに行かれたこともあるそうです．それに影響されて旅行や冒険がしたいと思い，卒業後オーストラリアにワーキングホリディで行きました．英語ができるようになりたかったので，カウンターのサンドイッチ屋さんとか町の工場とか，日本人のいない環境で仕事をしました．21歳の頃です．

トニー オーストラリアに行かれたときは，英語はできたのですか？　またエンジニアリングは？

下村 英語はまったくできませんでした．だから英語が必要なアルバイトを選んだのです．エンジニアリングは興味がなかったというか，まだコンピュータがあることさえ気づいていませんでした(笑)．

オーストラリアで12か月を過ごした後，また日本に戻ってバイトをし，今度はカナダに行きました．そのときは整体に興味があり，勉強したかったのですが，結局あまりよい手がかりが見つかりませんでした．せっかく大自然が豊富なカナダに来たのだから牧場で働こうと思い，牧場の仕事を探してハイジみたいな生活を3か月間過ごしました．その後，カナダでいろいろな人と出会い，会社をやめて海外の大学に留学していた人達に感化され，自分も海外の大学で勉強したいと考えました．

トニー ワーキングホリディを利用されたのですよね？

下村 そうです．その後日本に戻り，本格的にアメリカの大学に入るための準備をし，アメリカに来ました．オレゴン州の大学でした．整体治療の分野に興味があり，カイロプラクターをめざすため，その習得科目を勉強することにしていたのですが，医学の準備コースの英語がとても難しく，挫折しそうになりました．そのうちに国連などの国際的に貢献できる仕事をしたいと思い，その学科がある南カルフォルニアのサンディエゴの大学に編入しました．でも現実を調べてみると，なかなか国連などのそういう政府関係の仕事をするにはいろいろな制限があり，キャリアにつながるか疑問がありました．それで，国際ビジネスに専攻をまた変更しました．

トニー 夜学の先生の影響でオーストラリアへ，整体療法を求めてカナダへ，そしてアメリカですか．まさに放浪の旅ですね！

下村 いえいえ(笑)．この先にまだエンジニアリングに落ち着くまでのストーリーがあります！　卒業後，アメリカに残るための就労ビザのサポートを得るためには，一般的なビジネスよりもコンピュータの知識が非常にプラスになるとわかりました．それでMIS科[注1]に編入したのです．まったく未知の世界だったのですが，宿題のプログラミングに夢中になり，つい徹夜してしまったりして，コンピュータの分野が自分に向いているのだと気づきました．

☆ アメリカでの大学生活

トニー アメリカの大学生活はどうでした？

下村 はじめの頃は英語力が気になりますね．TOEFLのスコアは日本で準備していたのであったのですが，生きた英語での授業はまた違います．

トニー そうですよね，とくに文系を進んでいたらたくさん本を読まなくてはいけないし，論文やエッセイもたくさん書かなければならないし，クラスでも発表したりディベートできないといけないから，大学レベルの英語力はネイティブの人でも苦労しています．

下村 やっぱりそうですか……．もっとも困ったことは，グループプロジェクトですね．時間にルーズな人や，グループでミーティングしてもただ喋るだけでいっこうに仕事をしないフリーライダーと呼ばれる人などもいて，苦労しました．

トニー そうですよね，一見フレンドリーなアメリカ人達が，一緒に仕事をしてみると意外と違った側面を見せてくれますよね．

☆ シリコンバレーのスタートアップにいきなり入社……そしてレイオフ

トニー シリコンバレーには，スタートアップにいきなり入ったのですよね？

下村 エンジニアとして働くならシリコンバレーでと思い，在学中に最後の単位をインターンとして取ることに決め，卒業前にいきなりシリコンバレーに引っ越してきました．オンラインショッピングをやっているスタートアップの会社で，サーバサイドプログラミングとデータベースプログラミングなどの

注1：Management Information Systemの略，日本の情報管理化に若干似ているが，管理職系の人達にいかにITを利用するか，ビジネス面からアプローチをする分野．

Silicon Valley

H. Tony Chin

対談編

e-Commerceのバックエンドの構築に従事しました．大学でやっていたのはCやJavaだったのでPerlの経験はなかったのですが，インターンの間に必死で覚え，3か月後の卒業と同時に正式採用が決まりました．その1年後にシステムの移行があり，サーバをWindowsからLinux，データベースをSQLサーバからPostgreSQL，そして言語をPerlからPHPに置き換える大がかりなデータマイグレーションを学ぶことができました．のちに転職する際に，この経験に救われることになるのですが……．

トニー インターン制度を利用するのは，なかなかスマートなやり方ですね．現在の会社に移ったのは何がきっかけでしたか？

下村 その会社の経営状態が危うくなってきて，それに拍車をかけたのが9/11のテロでした．開発部の半分がレイオフになり，私もその対象となったのです．アメリカではよくあることのようですが，日本の雇用制度になれていた私にはかなりショックでした．インターネットバブルも弾けたし，テロ後だったので非常に難しい状況でしたが，経験と即戦力性を買われ，今の会社に移ることができました．

トニー それは，下村さんにスキルがあり，ネットワーク能力が高かったからでしょうね．多くの外国人エンジニアが帰国したり，アメリカ人もシリコンバレーを離れていますから．シリコンバレーで驚いたことは？

下村 サンディエゴや南カルフォルニアに比べるとアジア系の人々が多く，引っ越し先のサニーベールはとくにインド人が多く住んでいました．留学生の頃は自分の英語力をいつも気にしていたのですが，シリコンバレーでは英語力ではなく仕事で人を評価するのだと感じました．

☆ 一人何役もこなす

トニー 現在の職場ではどんな事を？

下村 現在の会社では，おもにB2B/B2Cシステムの構築における設計/導入/運用及びプロジェクトのマネージメントをやっています．今手がけているプロジェクトはFailure Analysis，日本語でいうと失敗原因解析システムの開発をしています．過去の失敗事例を分析し，体系化することにより同じミスを防ぐためのシステムです．

以前の会社では，一開発者として言われたものを作っていればよかったのですが，今の会社では，データベース設計から，プログラミング，導入サポートなど，一人で何役もこなしています(笑)．

この分野のビジネスはとても競争が激しいので，シンプルにサポートできて，いかに汎用性を高くしてコストを削減していくかが課題です．また，Web系の技術はドンドン変わっていくので，それに対して敏感に情報を集めたりと，常に勉強していかなければなりません．とくにアーキテクトの部分では，まだまだ納得のいく仕事ができていないですね．

下村朋子氏

トニー それはどういうことですか？

下村 大きい会社だと同僚や先輩の意見を聞いたり，各分野の専門の人に相談できますが，今は私が開発の責任者なので自分の考えた方法が最善なのか？と心配になるのです．まあ，細かい技術的な疑問は友人とかに聞いたりしています．たとえばDBチューニングのノウハウなどは，データベースの会社に勤めている友人に聞いたりしてヒントを得ています．

トニー ネットワーキングで知り合った人達ですか？

下村 まあ，そういう人達もいますね．後は，コミュニティカレッジなど，安価にさまざまなプログラミングのクラスを受けられる環境がそろっているので，それを利用します．そこで基本的なことは覚えられるし，友達の輪も広がりますよ．やっぱり自分から新しい情報を集めるテクニックとか，知らないことを教えてくれる人を周りに作るのは大事です．

トニー シリコンバレーらしいネットワークの活用術を実践していますね．今後は？

下村 バックエンドをやっていたので，もう少しユーザーに近い仕事をしてみたいですね．バックエンドの技術がどんどん簡易化されていく中で，インターフェースのデザインがますます重要視されてくると思います．それをうまく生かしたEラーニングや教育ソフトが面白いと思います．自分のペースで学んだりできるシステムが安価に供給されれば，場所を選ばずにいろいろなことが学べるし，たとえば体が不自由な人や家庭の主婦とかでもドンドン利用できると思うのです．Flashとかインタラクティブなものは学ぶ人を飽きさせないし，フィードバックがあると面白く，作るほうもクリエイティブな意欲が沸きます．

対談を終えて：

じつに背景がさまざまなエンジニアがいるとつくづく感じた対談だった．とくに放浪の旅の中から出てきたエンジニアリングの話が非常に新鮮だった．さまざまなことにトライした結果がエンジニアリングだった．一見大人しい日本人女性の雰囲気なのだが，下村氏はフットワークも軽く，行動力がかなりあると思う．かなりアグレッシブにネットワーキングを活用して，自分の技術的スキルを日々磨いている．

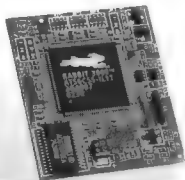
トニー・チン htchin@attglobal.net WinHawk Consulting

HARD WARE

●CPUモジュール

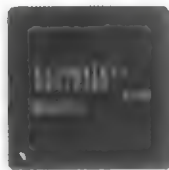
RCM3400 RabbitCore

- Z80/Z180を基本とした命令セットをもつCPU，Rabbit 3000を搭載したCPUモジュール．
- 小型サイズの基板（34×29mm）に，512Kフラッシュメモリ/512K SRAMまたは256Kフラッシュメモリ/256K SRAM，五つのシリアルポート，8チャネルのゲインを搭載．
- プログラム可能なアナログ入力を搭載した29.4MHzで走るLow-EMIのRabbit 3000ベースのCPUサブシステム．
- あらかじめ割り当てられたMAC IDを装備することで，Ethernetの利用が可能．

■ **東京電子販売（株）**
価格：下記へ問い合わせ
TEL：03-5350-6711　FAX：03-5350-6867

●16ビット1チップマイコン

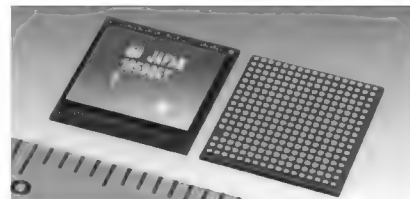
H8S/2168F

- PCサーバのファンコントロールなどに適した16ビット1チップマイコン．
- PCサーバの周辺ハードウェア用の共通インターフェースである，IPMI仕様バージョン1.5に準拠した各周辺機能を内蔵．
- 周辺ハードウェアの管理システムの高機能化と，遠隔管理を含めた高信頼化をコンパクトに実現．
- 通信インターフェースとして，シリアルインターフェース3チャネルに加え，I²Cバスインターフェースを6チャネル内蔵しており，BMCにつながるメモリ，外部補助基板，各種センサとの通信が可能．
- イベント機能付きPWMタイマを内蔵することにより，PCサーバに不可欠なファンの定速動作を実現することが可能．

■ **（株）ルネサス テクノロジ**
サンプル価格：¥1,700
TEL：03-5201-5276

●MPEG-4マルチコーデックLSI

Marvie3

- W-CDMA端末などに用いられるMPEG-4シンプルプロファイルの圧縮/伸張処理に対応．
- QVGA（320×240画素）サイズの液晶に対応し，MPEG-4規格に準拠した動画像の圧縮/伸張を低消費電力で実現．
- 3系統の同時圧縮/伸張により，TV電話を使いながら相手側画像の録画（圧縮2系統，伸張1系統）が可能．
- インターネットなどのコンテンツと，蓄積メディアデータを同時に再生しながら録画（圧縮1系統，伸張2系統）が可能．

■ **松下電器産業（株）**
サンプル価格：¥5,000
TEL：075-951-8151
E-mail：semiconpress@scd.mei.co.jp

●パワーマネジメントIC

TPS2370/TPS2383

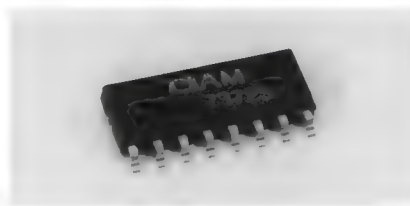
- 「TPS2370」はPower Over Ethernetを実現するパワーインターフェーススイッチ，「TPS2383」はパワーマネージャ．
- 「TPS2370」は，電源供給機器と負荷側電源機器間のインターフェースとして動作し，標準のEthernet用ツイストペアケーブルに接続された負荷側電源機器の検出，分類および低電圧ロックアウト，突入電流制限，FETスイッチの制御など，IEEE 802.3af規格に準拠するために必要な各機能を提供．導通抵抗0.3Ωの内蔵FETにより，最大の電源供給能力を提供し，システム内の発熱を最小に抑える．また，外付けの25kΩ検出抵抗を使用することにより，正確なPD検出機能を提供．
- 「TPS2383」は，最高8ポートまでのEthernetポートへの電源供給を個々に独立して管理する機能を備える．すべての動作は，I²Cシリアルインターフェース経由によるレジスタのリード/ライト動作によって制御される．

■ **日本テキサス・インスツルメンツ（株）**
価格：$1.25（TPS2370/1,000個時）
$7.00（TPS2383/1,000個時）
FAX：0120-81-0036
URL：http://www.tij.co.jp/pic/

●OPアンプ

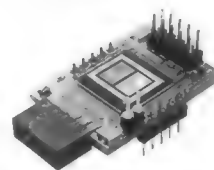
AM-7372S

- 高速，大電流出力のデュアル電圧帰還型OPアンプ．
- ±15V電源，±5V電源使用時の特性を規定した製品．
- 85dBのオープンループ利得の性能を維持して，ゲインバンド幅積は120MHz，スリュー率は3000V/μs，150mAの大電流出力性能を実現．
- 消費電力は，1アンプあたり6.5mA．
- 最高高調波ひずみは，－80dB/1kHz 2Vpp．
- 差動位相/利得は，0.01%/0.02°．
- 16ピンSOPパッケージで供給．

■ **デイテル（株）**
価格：¥440（10～24個時）
TEL：03-3779-1031　FAX：03-3779-1030

●FPGAコンフィグレーションモジュール

JtagLink JLA32M

- JTAG対応のFPGA自動コンフィグレーションモジュール．
- 100万ゲート規模のFPGA回路データの平均書き込み時間は，約25秒．
- 100万ゲート規模FPGAの自動コンフィグレーション時間は，約1秒．
- PCのパラレルポートに装着したまま，QuartusⅡ/iMPACTからユーザ基板への手動コンフィグレーションが可能．
- 自動コンフィグレーション実行前に，各FPGAのデバイスIDを照合．
- 32Mビットのフラッシュメモリを搭載．
- Altera/Xilinxの単一環境，および混在環境でのコンフィグレーションに対応し，JTAGチェーン中の非FPGAデバイスを自動スキップ．

■ **（株）DesignGateway**
価格：¥26,800
FAX：03-6644-1121
E-mail：info@dgway.com

●Ethernetモジュール

Digi Connect ME

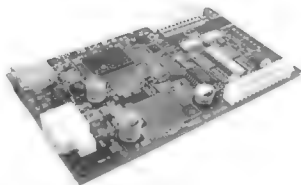
- 32ビットEthernet CPU「NS7520」を搭載した小型組み込みネットワークモジュール.
- 10/100Base-T Ethernetと高速なシリアルインターフェースをもち，製品基板上のシリアル信号に同製品を接続するだけで，ネットワーク化を実現.
- 18.29×18.67×36.57mmのコンパクトなサイズにより，従来Ethernet対応が困難であった小型機器へも対応.
- ネットワーク基本ソフトであるNET+ OSを搭載し，TCP/IP，Webサーバをはじめとする各種ネットワークアプリケーションを内蔵.
- 「NET+Works」統合開発キットにより，特有のアプリケーション開発やカスタマイズを行える.
- 「NS7520」は，ARM7をコアに，10/100 Base-T Ethernet MAC，13チャネルDMAコントローラ，メモリコントローラなどの主要周辺回路を1チップ化.

■ ネットシリコン ジャパン(株)
価格：下記へ問い合わせ
TEL：03-5428-0261　FAX：03-5428-0262
URL：http://www.netsilicon.co.jp/

●制御用USB I/Fコントロールユニット

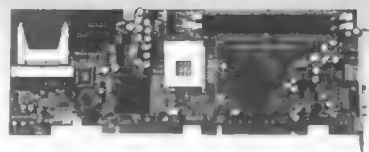
USB IO-8

- USBに対応した入出力制御ユニット.
- サイズが70×120mmのボードタイプで，省スペースでI/O機能の追加が可能.
- ブリンク機能，立ち上がり/立ち下りのラッチ機能などの関数が標準で装備されているため，プログラムの構築が簡単.
- ディップスイッチの使用により，最大127デバイスまでの増設が可能.
- 入出力はフォトカプラ入力8点，オープンコレクタ出力8点を用意.
- パソコンからのセルフパワーでも利用可能.
- USBバージョン1.1準拠で，フルスピード12Mbpsに対応.
- 使用周囲温度は0～40℃，保存周囲温度は0～50℃.

■ (株)エヌエスティー
価格：¥15,000
TEL：053-428-4823　FAX：053-428-4312

●シングルボードコンピュータ

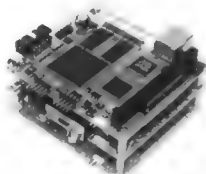
SAGP-845EV

- IEI社が開発した，AGPインターフェースのPIAGPシリーズのシングルボードコンピュータ(SBC).
- SBCのAGPバスの信号を，新しいPIAGPバスまで変換.
- オンボードVGA機能が要求を満足できない場合，内蔵された2x～8xAGPを通じ，高速の画像データバスを提供.
- PIAGPインターフェースのバックプレーンは，CPUスロット用にPIAGPおよびPICMGを揃え，拡張はAGPとPCIを標準インターフェースとしている.
- 従来のPICMG機能をサポートし，バージョンアップの際には，PIAGPのCPUカードにも変更可能.

■ エーワン(株)
価格：下記へ問い合わせ
TEL：0568-85-8511　FAX：0568-85-8501

●SH-4 CPUボード

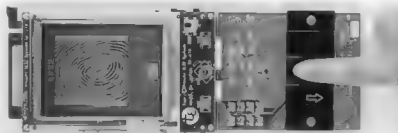
MS104-SH4

- PC/104標準バスに準拠し，組み込みアプリケーションの構築に適するCPUボード.
- PC/104規格のモジュールを増設することで，機能拡張が可能.
- ルネサステクノロジ社製のSH7750R(240MHz)を搭載.
- 標準OSにLinux(カーネルバージョン2.4)を採用.
- 16Mバイトのフラッシュメモリ，32MバイトのSDRAMを標準搭載.
- 10/100Base-TXを1ポート装備.
- CompactFlash(Type I)を1ポート装備.
- シリアルI/Fは，EIA-232ドライバを2ポート搭載し，最大伝送速度は921.6kbps.

■ (株)アルファプロジェクト
価格：¥42,800
TEL：053-464-2166FAX：053-464-3737
E-mail：sales@apnet.co.jp

●指紋認証システム開発キット

Finger Attestor for T-Engine

- 静電容量方式スイープ型センサを用いた，指紋認証システム.
- サイズは192ドット(17.0mm)×8ライン(3.0mm)，解像度は363dpi.
- 携帯電話や携帯端末などの小型組込み機器への最適化を実現.
- 用途に応じてセキュリティレベルの変更が可能.
- カーソル，ポインティングデバイスとしても使えるデュアル機能.
- 動作電圧は2.5～3.3V，消費電流は5mA.

■ パーソナルメディア(株)
価格：¥140,000
TEL：03-5702-7858
E-mail：te-sales@personal-media.co.jp

●CPUボード開発キット

Embedded Linux Reference Kit for MontaVista

- Geode CPUボード上でのMontaVista Linuxの性能を評価するためのスタータキット.
- CPUボード，I/Oボード×2，OS，各種ドライバ(拡張キット)および回路図で構成.
- プリビルドされているカーネル，ルートファイルをインストールするだけで，Geode CPU，I/Oボード上での基本的な実行環境が構築可能.
- 拡張キットをアドオンすることにより，CPU，I/Oボードが搭載するほとんどのI/Oの使用が可能になる.
- 同梱のハードウェアはピノー社製で，米国ナショナルセミコンダクタ社x86互換CPU Geodeを搭載している.

■ 加賀電子(株)
価格：¥500,000
TEL：03-3942-6295　FAX：03-3942-7399

■弊誌では新製品に関するニュースリリースを募集しております.
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　(編集部)

HARD WARE

●コンパクトコントローラ

AS-C1000

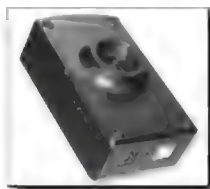
• Altera社のFPGAデバイス「Cyclone」を採用し，Niosプロセッサを組み込んだ可塑性の高いコンパクトなシステムを構築.
• キー入力，表示部一体型モータコントロール基板は，総計50桁の7セグメント表示とスイッチ入力，4軸のモータを制御.
• FPGAの構成を変えることで，同一の基板で4種類の機能を実現.
• 個々のIC間の電気特性を考慮する必要がなく，電気特性のミスマッチによる不具合の発生がなくなる.
• ロジックデバイスを1個のデバイスに集約できるため，配線パターンを削減でき，ノイズなどの影響を最小限に抑えることができる.
• 目的の仕様にフィットしたシステムをコンパクトに実現可能.
• 仕様変更にともなう回路変更の必要が発生しても，ICやユニットを取り替える必要がない.

■ シーク電子工業（株）
価格： 下記へ問い合わせ
TEL ： 075-621-6792　FAX ： 075-621-6679
URL： http://www.seekgr.com/

●USB2ビデオカメラ

Lu100シリーズ

• Limenera社が開発したメガピクセルカメラで，広範囲なアプリケーションに対応できるようデザインされている.
• カラーまたはモノクローム，プログレッシブスキャン，1.3MピクセルCMOSセンサを搭載.
• 1280 × 1024ピクセル分解能，オンボード処理で工業用，科学用のアプリケーションに高品質イメージを提供.
• ライブで連続Videoイメージや，スチールイメージを非圧縮で，USB2インターフェースを介して取り込むことができる.
• フレームグラバーボードは不要.

■ （株）アルゴ
価格： ¥128,000
TEL ： 06-6339-3366　FAX ： 06-6339-3365
E-mail ： argo@argocorp.com

●ディジタルオシロスコープ

DL1640/DL1640L DC電源モデル

• 「DL1640」は8Mワード，「DL1640L」は32Mワードメモリのディジタルオシロスコープ.
• 別売の充電式バッテリボックスの併用で，約2時間のバッテリ駆動が可能.
• ACアダプタ機能を内蔵し，ACラインから電源供給を受けられる.
• 停電や電圧低下などのライン電源トラブル時には，自動的にバッテリ駆動に切り替わり，測定の継続が可能.
• 4チャネル使用時でも最大で32Mワードのデータを捕捉することが可能なため，長時間の測定でもサンプリングスピードを犠牲にすることなく，現象の全体と詳細を一度に観察することが可能.

■ 横河電機（株）
価格： ¥680,000（DL1640）
¥950,000（DL1640L）
TEL ： 0120-137-046　FAX ： 0422-52-6624

●プロトコルテスタ

K15シリーズ

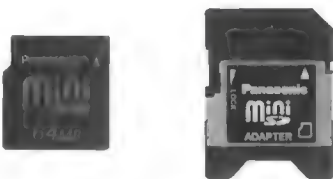
• UMTS/W-CDMA，GPRS，GSM，CDMA 2000，cdmaOne，EDGEなど最新の3Gネットワーク用プラットホームをサポートしたプロトコルテスタ.
• 3Gネットワークの運用/保守で，総合的でリアルタイムな解析が可能.
• 高度な解析項目にも柔軟に対応する，3G用エキスパートシステムを搭載.
• リアルタイムマルチインターフェースとして，コールトレース機能を搭載し，コールトレースが動作中でもストリームデータをディスクに収集可能.
• 自動パラメータ設定機能などで，3Gネットワークの複雑な解析を自動化し，ネットワークのダウンタイムを大幅に削減.
• データ取り込みとリアルタイムアプリケーションの同時動作を可能にし，より強力な解析を実現.

■ 日本テクトロニクス（株）
価格： ¥12,000,000～
TEL ： 03-3448-3010　FAX ： 0120-046-011

●メモリカード

miniSDカード

• 「RP-SS032BJ1K」は32Mバイト，「RP-SS064BJ1K」は64Mバイトの記憶容量を持ち，静止画像記録枚数は約31枚と64枚，音楽データ録音時間は約40分と80分，平均転送レートは2Mバイト/s.
• 表面積は現行のSDカード比56%のコンパクトサイズで，SDメモリカードと同様にSDMI規格準拠の高速アクセスおよび高度な著作権保護機能を装備.
• 付属のminiSDアダプタを使用することで，現行のSDメモリカード応用商品にも使用が可能.

■ 松下電器産業（株）
価格： オープン価格
TEL ： 0120-878-365

● Linuxサーバ

L-Box

• ブロードバンドに対応し情報端末の制御を可能とする，小型Linuxサーバ.
• 119 × 68 × 98mm，重さ267gと小型，軽量で，Ethernet，RS-232-C，PCMCIAカード，無線LANカードなど各種インターフェース端子を装備.
• ADSLなどのブロードバンドに対応するゲートウェイ機能と，接続している各種情報端末を不正なアクセスから守るファイアウォール機能を有する.
• IPv4/IPv6に対応.
• 低消費電力設計で発熱量を抑え，安定した常時利用が可能.
• OSにはオープンソースのLinuxを採用し，周辺機器に合わせたOSの拡張やアプリケーションソフトの開発が可能.
• モバイルSOHO，ホームセキュリティ，アメニティコントロール，ネット家電，遠隔監視などの利用モデルに適する.

■ NTTコムウェア（株）
価格： ¥49,800
TEL ： 03-5463-5779
E-mail ： kouhou@nttcom.co.jp

●ITRON仕様OS

TOPPERS/IDL-Pro

- 「TOPPERSプロジェクト」で開発された，TOPPERS/IDLカーネルに，TCP/IPプロトコルとFATファイルシステムが移植されており，ネットワークやファイルシステムの利用が可能．
- オプションで，さまざまなミドルウェアを用意．
- 目的別にミドルウェアを組み合わせ，標準ハードウェアにインテグレーションした各種プロトタイプキットを提供．

■（株）エーアイコーポレーション
価格：下記へ問い合わせ
TEL：03-3493-7981　FAX：03-3493-7993
E-mail：sales@aicp.co.jp

●リアルタイムサブシステム

Hyper Kernel

- 米国ネマトロン社が開発した，Windows環境下でリアルタイムオペレーションを可能とするリアルタイムサブシステム．
- Windowsと共存して動作するが，完全な独立実行環境を有する．
- スレッドは同製品環境下で実行されるため，リアルタイムアプリケーションと非リアルタイムアプリケーションを同時に実行することが可能．
- ラウンドロビン法を用いた独自のスケジューラにより，Windowsのスケジュールを管理．
- Windowsとのデータの受け渡しは，シェアードメモリを介して行われ，メモリアクセスは用意された関数によって実現．
- ドライバ開発キットなどは不要．
- Windows HALの変更は不要．

■アークシステムズ（株）
価格：下記へ問い合わせ
TEL：03-3847-0216　FAX：03-3847-0235

●開発ツール

Borland C#Builder for the Microsoft .NET Framework日本語版

- NET Windows Forms，Web Forms，ASP.NET，ADO.NETを含め，Microsoft .NET Frameworkに完全に対応しており，デスクトップ，社内業務，Webアプリケーション，Webサービス，サーバコンポーネントなどの開発が可能．
- 同社のCORBA技術に基づくBorland Janevaにより，ブリッジ技術に頼ることなくJ2EEとCORBAアプリケーションを高速に統合可能．
- BDP（Borland Data Provider）for ADO.NETは，.NETデータタイプに完全対応した標準準拠のデータアクセスコンポーネント．プログラムを変更することなく，他のデータベースに移行することができ，Microsoft SQL Server，Oracle，IBM DB2，Borland InterBaseなどの主要なデータベースを.NETアプリケーションで利用することが可能．

■ボーランド（株）
価格：¥10,000～¥360,000
TEL：03-5350-9380　FAX：03-5350-9369
URL：http://www.borland.co.jp/

●リアルタイムJavaバーチャルマシン

JTime

- 米国タイムシス社が開発した，リアルタイムJava仕様に完全対応した組み込みJVMとその開発環境である「JTime SDK」．
- サンマイクロシステムズ社が認証した，完全にパフォーマンス予測可能なJVMであり，あらゆる組み込みシステムにおいてリアルタイムオペレーションを実現．
- JVMとして「TimeSys Linux SDK」のツールとして提供され，「JTime SDK」は同社のリアルタイムLinuxパッケージである「TimeSys Linux/Real-Time」と組み合わせることで，組み込みリアルタイムシステムに最適なパッケージとして提供．
- 「JTime」と「TimeSys Linux/Real-Time」のパッケージには，統合性の高いJava/Linuxの開発環境およびランタイム環境が含まれており，GNUツールチェイン，ルートファイルシステム，全てのLinuxパッケージ，ユーティリティおよびライブラリを提供．

■（株）日新システムズ
価格：下記へ問い合わせ
TEL：075-344-7977
E-mail：info@co-nss.co.jp

●Linux版ブラウザ開発キット

NetFront v3.1 SDK for Linux

- 情報家電向けブラウザを，Linux上に短期間で移植，カスタマイズが可能なSDK開発キット．
- Linuxディストリビューションとして，モンタビスタソフトウエア社の「MontaVista Linux」に対応．
- GUI環境として，「MontaVista Graphics/GTK+」，トロルテック社の「Qt/Embedded」および「Qtopia」をサポート．
- ウィンドウシステムAPI「AWS」により，X Window SystemやQt/Embeddedのウィンドウシステム上で，NetFrontのブラウザ機能を実現．
- Linux上でポピュラーな「Glade」「GTK+」「Qtopia」などの開発環境を使用して，ユーザーインターフェースのカスタマイズやブラウザアプリケーションの開発が可能．
- プラグインオプションとして，「Flash 6（Macromedia）」「Helix DNA Client（Real Player）」「Adobe Reader for NetFront（Adobe Systems）」などの提供を予定．

■（株）ACCESS
価格：¥5,000,000～
TEL：03-5259-3685　FAX：03-5259-3684
E-mail：prinfo@access.co.jp

●DSPアプリケーション開発用コンパイラー

SuperH RISC engine C/C++コンパイラ Ver.8.0

- 積和演算や行列演算などDSP特有の演算命令をC言語で記述する「DSP-C拡張言語仕様」をサポートしているため，DSPの基本的な知識でC言語による平易な記述でプログラムを作成でき，デバッグ，保守を容易にする．
- DSPアプリケーションの開発期間を，アセンブラ言語で作成する場合と比較して，同社比で約1/5以下に短縮可能．
- これまでCPUで行っていた画像や音声などのデータ処理をDSPの処理とすることが容易であり，処理性能の向上を実現．
- DSP性能を発揮するための最適化したコードの生成により，CPU処理と比較して2～4倍の処理性能向上が可能．

■（株）ルネサス テクノロジ
価格：¥198,000～¥398,000
TEL：03-5201-5022

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　（編集部）

SOFT WARE

●携帯情報端末向けプラットホーム

Windows Mobile 2003 software for Pocket PC

- 無線LANゼロコンフィグレーション機能により，ネットワークを自動検知し，接続設定を行うことが可能.
- Bluetoothをネイティブサポートしており，多種多様な機器との通信が可能.
- 同期ソフト「Microsoft ActiveSync 3.7」により，予定表，メール，連絡先などMicrosoft Outlookの最新データの効率的な取得が可能.
- Exchange Server 2003とは，デスクトップPCを介することなく，直接データの同期が可能となる.
- 画像閲覧ソフト「Pictures」により，ディジタルカメラなどで撮影した写真画像の保存，編集，表示が可能.
- Windows Media 9シリーズのCODECをサポートしており，デスクトップPCのWindows Media 9シリーズの音声やビデオコンテンツの閲覧が可能.

■ マイクロソフト（株）
価格：下記へ問い合わせ
TEL：03-5454-2300

●マルチタスクデバッグ環境

HI ApplicationEngine for T-Engine

- Windows環境でマルチタスクプログラムのデバッグを可能にする，T-Engine用の開発環境.
- ルネサステクノロジ社製のSuperH純正CコンパイラおよびGNU開発環境に対応.
- ファイルシステムおよびグラフィックシステムを標準で搭載しており，アプリケーション開発が容易.
- T-Kernel BASIC版に対応しており，ITRONからT-Kernelへの移行に適する.

■パーソナルメディア（株）
価格：¥98,000
TEL：03-5702-7858　FAX：03-5702-7857
E-mail：te-sales@persnal-media.co.jp
URL：http://www.personal-media.co.jp/te/

●携帯音楽プレーヤ用システムIP

システムIP

- 次世代音楽圧縮技術Ogg Vorbisに対応した携帯音楽プレーヤ用LSI向けシステムIP.
- Ogg Vorbis携帯音楽プレーヤシステムの構築に必要な「ハードウェアIP」と「ソフトウェアIP」の統合パッケージで，追加のハードウェアやファームウェアは不要.
- 「ハードウェアIP」には，独自設計のCPUとDSPを内蔵する「マルチコアアーキテクチャ」を採用し，メモリカードインターフェース，外部メモリインターフェース，液晶表示，キー入力などの各種周辺デバイスの制御回路を統合. 外付け部品としては，メモリやD-Aコンバータなどの最小限の部品点数で済む.

■ ファインアーク（株）
価格：下記へ問い合わせ
TEL：03-5531-0211　FAX：03-5531-0205
E-mail：pr@finearch.com
URL：http://www.finearch.com/

●デバイスドライバ自動生成ツール

IAR MakeApp for ルネサスH8/SH

- IARシステムズ社が開発した，ルネサステクノロジ社製H8およびSH用のデバイスドライバ自動生成ツール.
- 新H8シリーズおよび3種類の新SH-2デバイスをサポート.
- 使用/不使用のポートピンの最新状態をカラーコードで示し，グラフィカルなピン使用表示が可能.
- ポイントアンドクリック動作だけで，周辺モジュールのデバイスドライバ（初期化，ランタイム，割り込みハンドラ）のCソースコードを自動生成.
- Project Explorerで，デバイスドライバ関数や割り込みハンドラを含めプロジェクト状態の概観を表示.
- 最適コードジェネレータが，効率的でテストされたソースコードを生成し，ルールチェッカと特殊機能レジスタ値の自動計算で，開発期間とデバッグ作業を削減.
- いつでもリソース使用状況がわかる視覚的な概観表示機能を搭載.

■（株）プロトン ソフトボート事業部
価格：下記へ問い合わせ
TEL：03-5337-6431　FAX：03-5337-6130

●ユビキタスデータベース

Encirq 3e

- 米国エンサーク社が開発した，組み込みシステムにおいてデータベースの設計を容易に行える，組み込み用データフロー管理ソフトウェア.
- データソースを抽象化することでデータフロー管理を行うため，開発者はネットワーク，入出力装置，補助記憶装置およびセンサなどのデータの入出力に関して考慮する必要がない.
- 設計したデータベースは，PC上でコマンドを入力しながら対話形式で行うか，自動生成されるC言語ソースコードをビルドした上で行うかのいずれかの検証方法を選択できる.
- Oracle PL/SQLをベースとした，組み込み用に特化した言語「Encirq PL」を採用.
- 50～75Kバイトという，コンパクトなコードサイズを実現.
- CPUやOSに非依存.
- SQLをサポートし，データ格納先は複数指定可能.

■（株）エーアイコーポレーション
価格：下記へ問い合わせ
TEL：03-3493-7981　FAX：03-3493-7993
E-mail：sales@aicp.co.jp

●開発ツール

インテル VTune エンタープライズ・アナライザ 2.0 .NET エディション

- インテル社が開発した，Webアプリケーションのコードを解析して，パフォーマンス上のボトルネックとなっている部分を見つけ出す解析ツール.
- スクリプト，ビジネスオブジェクトおよびデータベースクエリを含むオブジェクトレベルの解析と同様に，多階層のトランザクションレベルの分析を行える.
- マシン間の応答時間をアクティベーションレベルで監視できる.
- マシン間でのオブジェクト呼び出し状況を視覚的に表示するアプリケーションフロー分析機能を搭載.
- アプリケーションコンポーネントの応答時間とパフォーマンスメトリックスの分析機能を向上.
- マルチ層を対象にして解析を行った後，VTuneパフォーマンスアナライザを使用して，個々のシステムレベルでの分析が可能.

■ エクセルソフト（株）
価格：下記へ問い合わせ
TEL：03-5440-7875　FAX：03-5440-7876
E-mail：intel@xlsoft.com
URL：http://www.xlsoft.com/intel/

SOFT WARE

●多機能ソフトフォン

ViBro GATE シリーズ SIP ソフトフォン/SIP クライアントソフト開発キット

- 「SIP ソフトフォン」は，音声会話に加えて，PB キー入力，テキスト文字列の転送，ページアドレス入力による Web ブラウズ，マウス入力による画面への描き込みなどを可能にした，PDA/PC 用多機能ソフトフォン．
- 「SIP クライアントソフト開発キット」は，SIP ソフトフォンを短期間で開発可能にする開発キットで，SDK では，発呼，着信，音声データ入出力，DTMP/TEXT CODEC 入出力などの機能をサポート．
- 音声パケット損失時でも補正によりノイズが入らず，ステレオ CODEC に対応しているため，回線帯域，音声内容に対して最適なデータ形式を選択することが可能．

■（株）オーエスアイ・プラス
価格：オープン価格
TEL：03-3794-8408 FAX：03-3760-4800
E-mail：sales@osiplus.co.jp

●AES 暗号プログラム

iotaCRYPT/AES

- AES 暗号アルゴリズムを基に，独自の実装方法を用いて AES 暗号化と復号化を，高速かつ省メモリで実現した，C 言語版と Java 言語版の暗号プログラムをライブラリおよびソースコードで提供．
- NIST の次世代共通鍵暗号標準化プロジェクトで採択され，FIPS として発行された，従来の DES 暗号に代わり現時点で最高水準の強度をもつ暗号アルゴリズムを採用．
- C スタティックライブラリは 3K ～ 15K バイト，Java クラスライブラリは 2.5K バイトの ROM サイズを実現．
- 速度性能は，オリジナル AES プログラムの 30 倍以上の高速化を実現．
- ソフトウェアのみで実現しているため，特殊なハードウェアは不要．
- ライブラリだけでなく，ソースコードで提供することにより，組み込み用途に対して柔軟な対応が可能．
- 導入セミナーとサポートサービスを用意．

■（株）デンソークリエイト
価格：¥100,000 ～
TEL：03-3780-7081
E-mail：info@dcinc.co.jp

●セキュリティ対策ソフト

インターネットウィルスプロテクター V4

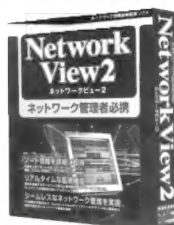
- Windows，マクロ，Java 用の 3 種類のエンジンを搭載し，各種ウィルスに最適化されたエンジンを使用することで，ウィルスの検出/駆除が可能で，未知のウィルスに対する検出機能も搭載．
- エクスプローラや Outlook 用のウィルス検出/駆除プラグイン機能を搭載．
- ウィルス感染の可能性がある経路をリアルタイムに監視．
- FTP，HTTP プロトコルをサポートし，ファイアウォールやプロキシ機能を利用している場合でもアップデートが可能．
- ブートウィルス感染などが原因で，システムが起動できない場合，復旧ディスクを作成してブートセクタの復旧が可能．
- スパムメールフィルタリング機能が搭載されており，発信者/件名/メッセージのヘッダにフィルタリング用のキーワードを登録することにより，迷惑メールの遮断が可能．
- 大容量ハードディスクをバックアップするために，記録型の DVD メディアに対応し，「B 's Recorder GOLD5」のエンジンを搭載．

■（株）ライフボート
価格：下記へ問い合わせ
TEL：03-3265-1250 FAX：03-3265-1251

●ネットワークモニタリングソフト

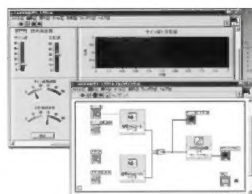
NetworkView2

- スイスのネットワークビューソフトウェア社が開発した，ネットワーク上の DNS，SNMP，ポートおよび MAC アドレス情報からノードを探索し，それらの情報をもとに，部内 LAN 管理だけではなく，ネットワークの保守，管理事業にも活用可能なネットワーク接続図と一覧表を作成．
- 5,000 以上のハードウェアメーカーとデバイスに関する OID データベースを標準で搭載し，探索したノードから得られた情報を基にある程度の機種特定が可能．
- OID データベースの新規追加や編集が可能．
- MIB（Management information base）ブラウザにより，相手先の SNMP エージェントよりさらに詳細なノード設定の確認が可能．

■ キヤノンシステムソリューションズ（株）
価格：¥19,800
¥12,800
（ダウンロード標準版）
TEL：03-5730-7198
E-mail：
nv-info@canon-sol.co.jp
URL：http://canon-sol.jp/

●グラフィカル開発環境

LabVIEW 7 EXPRESS 日本語版

- 一般的な計測/テストアプリケーションを構成可能なダイアログを利用して短時間で作成できる「Express VI」を搭載．
- 全部で 38 種類の「Express VI」は，高度な計算機能を手軽に利用でき，データ集録からファイル I/O 信号解析まで，幅広いタスクの開発効率を向上させることができる．
- 再設計された NI-DAQ ドライバフレームワーク，およびデータ集録と計測器制御向けの二つの新機能「DAQ アシスタント」および「計測器 I/O アシスタント」を装備．

■ 日本ナショナルインスツルメンツ（株）
価格：¥165,000（ベースパッケージ）
¥315,000（開発システム）
¥540,000（プロフェッショナル開発システム）
TEL：03-5472-2970 FAX：03-5472-2977
E-mail：prjapan@ni.com

●ASP.NET 学習ソフトウェア

Learn!ASP.NET

- .NET 戦略における Web アプリケーションの作成について，実用的な技術の習得を目的とした学習ソフトウェア．
- 「KeyStone Learn!」シリーズのローカライズ版．
- ASP モデル以前の技術を復習した上で，ASP.NET の各種コントロールの操作方法，ADO.NET によるデータベース操作/管理，XML ドキュメントの取り扱い，アプリケーションの運用技術について学習．
- 動画やプレゼンテーション画面を多用した講義をセミナ感覚で視聴でき，自由な時間に自己のペースで，繰り返し学習が可能．
- Visual Basic .NET および C# のサンプルプログラムが付属．

■ グレープシティ（株）
予定価格：¥58,000
TEL：022-777-8211 FAX：022-777-8233
E-mail：sales@grapecity.com

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨 1-14-2 Interface 編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp （編集部）

海外・国内イベント/セミナー情報

INFORMATION

海外イベント

9/15-18 **Embedded Systems Conference Boston**
Hynes Convention Center Boston, MA, USA
CMP Media LLC.
http://www.cmp.com/eventcal/

9/15-18 **TECHXNY/PC EXPO**
Jacob K.Javits Convention Center, NY, USA
CMP Media LLC.
http://www.techxny.com/home.cfm

9/16-18 **COMDEX Canada 2003**
Metro Tronto Convention Centre, Toronto, Ontario, Canada
MediaLive International Inc.
http://www.comdex.com/canada/

9/18-21 **CeBIT Asia**
Shanghai New International Expo Centre, China
Hannover Fairs China Ltd.
http://www.cebit-asia.com/

9/30-10/1 **Embedded Systems Conference Asia**
Lakeshore Hotel, Hsinchu, Taiwan
CMP Media LLC.
http://www.english.escasiaexpo.com/

9/30-10/2 **International Test Conference 2003**
Charlotte Convention Center, Charlotte, NC, USA
ITC Office
http://www.itctestweek.org/

9/30-10/2 **Communications Design Conference**
San Jose Convention Center, San Jose, CA, USA
CMP Media LLC.
http://cmp.iconvention.com/cdc/V40/

国内イベント

8/28-29 **mobidec 2003**
青山ダイヤモンドホール(東京都渋谷区)
モビデック事務局
http://www.shoeisha.com/event/mobidec/

9/1 **802.11 PLANET Conference&Expo Japan 2003**
渋谷マークシティ(東京都渋谷区)
IDGジャパン/米国Jupitermedia社
http://www.idg.co.jp/expo/j802.11/

9/10-12 **第5回 自動認識総合展**
東京国際展示場(東京ビッグサイト，東京都江東区)
(社)日本自動認識システム協会
http://www.autoid-expo.com/

9/17-20 **WPC EXPO 2003**
日本コンベンションセンター(幕張メッセ，千葉県千葉市)
日経BP社
http://expo.nikkeibp.co.jp/wpc/ja/

10/7-11 **CEATEC JAPAN 2003**
日本コンベンションセンター(幕張メッセ，千葉県千葉市)
CEATEC JAPAN運営事務局
http://www.ceatec.com/index.html

10/29-30 **Internet&Mobile 2003**
マイドームおおさか(大阪府大阪市)
(社)日本能率協会
http://www.jma.or.jp/im/

10/29-31 **DATABASE 2003 TOKYO**
東京国際フォーラム(東京都千代田区)
財団法人データベース振興センター(DPC)
日本データベース協会(DINA)
http://www.dbtokyo.com/

開催日，イベント名，開催地，問い合わせ先の順

セミナー情報

システムコールで学ぶLinux
開催日時　：8月28日(木)
開催場所　：ディーアイエステクノサービス研修室(東京都文京区)
受講料　　：46,000円
問い合わせ先：(株)エイチアイICP事業部，☎(03)3719-8155，FAX(03)3793-5109
http://icp.hicorp.co.jp/seminar/linux/linuxsystemcall.asp

非接触ICカード/RFタグの基礎
開催日時　：8月28日(木)
開催場所　：中央大学駿河台記念館(東京都千代田区)
受講料　　：52,500円(1口で1社3名まで受講可)
問い合わせ先：(株)トリケップス，☎(03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/c030828b.htm

すぐできる「組込みLinux開発」
開催日時　：8月28日(木)～8月29日(金)
開催場所　：(株)ソリトンシステムズ本社7Fセミナールーム
受講料　　：160,000円
問い合わせ先：(株)ソリトンシステムズ トレーニング担当，
☎(03)5360-3819，training@soliton.co.jp
http://sbo.soliton.co.jp/products/embedded_sbc/LinuxSeminarHP2-3.htm

ネットワーク/IPv6(ネットワーク構築)
開催日時　：9月9日(火)～9月10日(水)
開催場所　：高度ポリテクセンター(千葉県千葉市)
受講料　　：35,000円
問い合わせ先：雇用・能力開発機構 高度ポリテクセンター事業課，☎(043)296-2582
http://www.apc.ehdo.go.jp/

ホームネットワーク最新動向とUPnPの応用
開催日時　：9月11日(木)
開催場所　：オームビル(東京都千代田区)
受講料　　：52,700円
問い合わせ先：(株)トリケップス，☎(03)3294-2547，FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/030911n.htm

エクストリーム・プログラミングXP導入のためのワークショップ
開催日時　：9月16日(火)～9月17日(水)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：78,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/23/23_142.htm

情報セキュリティ技術基礎
開催日時　：9月17日(水)
開催場所　：NRI大手町ラーニングセンター(東京都千代田区)
受講料　　：47,250円
問い合わせ先：(株)エイチアイICP事業部，☎(03)3719-8155，FAX(03)3793-5109
http://icp.hicorp.co.jp/seminar/nrisecu/secu_kiso.asp

Windowsセキュリティ －攻撃手法の理解と対策－
開催日時　：9月19日(金)
開催場所　：NRI大手町ラーニングセンター(東京都千代田区)
受講料　　：47,250円
問い合わせ先：(株)エイチアイICP事業部，☎(03)3719-8155，FAX(03)3793-5109
http://icp.hicorp.co.jp/seminar/nrisecu/secu_win.asp

最新CCDイメージセンサの特性と技術～CCDの基礎と応用技術～
開催日時　：9月19日(金)～9月20日(土)
開催場所　：CQ出版セミナールーム
受講料　　：25,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，FAX(03)5395-1255

組み込みXML入門セミナー
開催日時　：9月26日(金)
開催場所　：クイーンズタワーB 7Fクイーンズフォーラム会議室(横浜市西区)
受講料　　：無料
問い合わせ先：(株)グレープシステム基本ソフトウェア事業部セミナー係，
☎(045)222-3761，FAX(045)222-3759
http://www.grape.co.jp/seminar.html

ビデオ信号の処理回路技術
開催日時　：9月27日(土)
開催場所　：CQ出版セミナールーム
受講料　　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局，☎(03)5395-2125，
FAX(03)5395-1255

PC実習!!Javaによるプロセス指向と並列プログラミング入門
開催日時　：9月29日(月)～9月30日(火)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　　：68,000円
問い合わせ先：(株)ソフト・リサーチ・センター，☎(03)5272-6071
http://www.src-j.com/seminar_no/23/23_253.htm

Linux GUIプログラミング
開催日時　：10月6日(月)
開催場所　：ディーアイエステクノサービス研修室(東京都文京区)
受講料　　：46,000円
問い合わせ先：(株)エイチアイICP事業部，☎(03)3719-8155，FAX(03)3793-5109
http://icp.hicorp.co.jp/seminar/linux/clinuxgui.asp

日程はすべて予定です．問い合わせ先にご確認のうえ，お出かけください．

読者の広場

Interfaceへの声

2003年8月号特集「現代コンピュータ技術の基礎」に関して

▷特集の用語解説はとても参考になりました．頭文字をとった略語などは，何気なく使っていますが，人に説明するとき間違った英語で言ってしまったり，説明できないときもあります．また略語の意味を調べるときに役立つと思うので，今月号は別保管したいと思います．（福沢陽介）

その他

「プログラミングの要」にあったナンセンスとしか見えないような規約の採用に関する記事には納得させられるものが多くありました．ただ本質的にすべての担当者に理解してもらえれば不自然な規約は必要なくなるのだが，そのための時間や教育をするリソースが取れず，とりあえず乗りきるために規定されている現場が多いのかもしれないとも感じました．（玉出のタマ）

▷「メモリプロファイリングツールを開発する」は，いわれてみれば「そうだ」と思えるほど興味がわく話題だった．もともとデータベースのプロファイリングの延長で作られているようだが，応用分野はそれ以外にもたくさんあると思われる．プログラムの概要の説明はあるだろうが，分野が異なる具体的な応用例もいくつか紹介されることを期待する．（N）

アンケートの結果

興味のあった記事（2003年8月号で実施）

①第1章 組み込みシステム開発の基礎知識
②第3章 基礎からの計算科学・工学――シミュレーション
③第2章 組み込み分野へのBSDの適用
④第7章 作りながら学ぶコンピュータシステム技術
⑤第5章 徹底解説！ ARMプロセッサ
⑥第6章 多国語文字コード処理＆国際化の基礎と実際
⑦日本語が使えるUMLツール最新比較
⑧フリーソフトウェア徹底活用講座（第11回）
⑨フジワラヒロタツの現場検証（第71回）
⑩第4章 データベース活用技術の徹底研究
⑪第10章 解説！ USB徹底活用技法
⑫USB Compliance Testの概要
⑬第8章 ワイヤレスネットワーク技術入門
⑭第9章 ICカード技術の基礎と応用
⑮別冊付録 エンジニアに役立つ数式集
⑯SH-4 Linuxの割り込み処理とPCIの割り込み共有について
⑰開発技術者のためのアセンブラ入門（第20回）
⑱やり直しのための信号数学（第17回）
⑲メモリプロファイリングツールを開発する（基礎知識編）
⑳シニアエンジニアの技術草子（参拾之段）

特集担当デスクから

☆2号連続特集企画の第一弾は，パイプラインとスーパースカラに焦点を当て，実際に市場で使われている各種プロセッサの事例もまじえて，マイクロプロセッサについて解説しましたが，いかがだったでしょうか．
☆CPUコアのブロック図やパイプラインのステージ詳細など，公式には発表されていないプロセッサもあるため，一部筆者による推定の部分があります．間違いなどありましたら，編集部までご指摘いただけると助かります．
☆次号は，キャッシュやMMU，そして例外処理などについて，今月号同様に実際のプロセッサの事例をまじえて，徹底的に解説する予定です．ご期待ください．

読者プレゼント

●応募方法：本誌読者アンケートはがきに必要事項を記入のうえ，2003年9月30日（必着）までにご応募ください．なお当選者の発表は，発送をもってかえさせていただきます．

（1）Flashのための Swift3D マスターブック（1名）
田崎進一/大河原浩一 共著
ISBN4-274-06527-8（株）オーム社

（2）ゲーム開発のための物理シミュレーション入門（1名）
David M.Bourg 著　榊原一矢 監訳
ISBN4-274-06526-X（株）オーム社

次号予告『詳説マイクロプロセッサ――キャッシュ/MMU/例外』

2003年11月号は9月25日発売です

キャッシュ/仮想記憶/TLB/MMU/例外/割り込み/命令セットアーキテクチャ

2号連続特集の後編にあたる次号では，おもにキャッシュ，MMU，例外について解説する．

まずキャッシュの章では，ダイレクトマップやフルアソシアティブ，nウェイセットアソシアティブなどの代表的なキャッシュの方式や，その動作を詳細に解説する．MMUの章では，仮想記憶の概念，アドレス変換やTLBなどについて，また68系や86系のMMUの動作について詳しく解説する．例外の章では，ソフトウェア割り込みとハードウェア割り込みの違いや，各種プロセッサごとの割り込み処理アーキテクチャの違いなどを解説する．

さらに，CISCからRISCへ，命令セットアーキテクチャがどのように変化してきたかを，代表的なプロセッサを例として取り上げ，各種プロセッサの命令セットについて比較/考察を行う．

また高速化技術や高信頼性をサポートする機能について，最近話題のコンフィギュラブルプロセッサについても説明する．

編集後記

■「デザインウェーブ・テクノロジ・セミナー(SystemC&SystemVerilog)」が，本号発売直後の8/29に開催される．名称は姉妹誌のDWM誌の誌名からとっている．さて，「インターフェース・テクノロジ・セミナー」とすると，内容は何が良いと思われますか？　組み込みLinux？ロボティクス？デバドラ？通信？……　(洋)

■2号連続企画という「暴挙」を企てたのは，何を隠そうこの私です(笑)．今月号はもちろん，次号の担当も私です．そして実は，Interface増刊TECH I Vol.18の担当も私で，さらに駄目押しで本誌特集と同時進行！　終わるまで夏休みは取れそうにありません(涙)　ちなみに増刊もプロセッサの話なので，頭がごっちゃになってます!?(M)

■夏といえば発熱．最近の大容量HDDは7,200rpmばかりになり，低発熱の5,400rpm品を愛用していた私にはかなり辛いところ．遅くても低発熱品のHDDが良いという場合は，小容量で我慢するしかないのか……．仕方ないので，アルミケースやファンなどでの対症療法を検討しているところです．　(み)

■今月からインターフェースの担当になりました．トラ技からお引っ越しです．引っ越し直後なので「どうせ編集後記は来月号からだろ」と，たかをくくっていたら，「今月から書いてね～」との指令が！　トラ技校了直後なので完全にネタ切れ状態です．とりあえず，またこのスペースに文章をピッタリ収めることから始めましょうかね．　(@)

■インターネットのおかげで情報公開が進むのはよいことだが，悪用も多い．よく未成年犯罪者の氏名が掲示板に公開されて問題になる．昔は，口コミで一部の地域だけですんでいたものが，全国に拡大されたというわけだ．ミニコミなら影響は少なくても，マスコミになると大事になる．モラルが欠落していく時代に，よりモラルが必要になるというのも皮肉だ．　(Y)

■8月になって梅雨明け．昨年みたいに暑い日があまり無かったと思っていたら，なんと7月の日照時間の少なさが観測史上初だったようで……．「昔は四季がはっきりしていた」なんて話にもなりました．小学生の時には，夏休みが始まる頃はもう夏真っ盛りでしたもの．やはりこれも環境破壊などのせいなのでしょうね．(Y2)

■最近，自宅には高校を卒業する娘へのダイレクトメールが，山のように配達されてきます．配達されたDMのほとんどがごみとして廃棄されますが，分別処理するのが大変です．反面，学校の学生確保と運営も大変な時期に入っている事を実感します．ただし，知名度の高い学校からは，それほど届きません．　(S)

■話題となった「バウリンガル」に続き，待望の猫語翻訳機「ミャウリンガル」が11月ごろ発売されるそうです．訳してもらえると分かれば，猫もさぞかし言いたいことがたくさんあることでしょう．しかし飼い主にとっては，ぜひとも人間の言葉を猫に伝えてもらいたいものですが．　(e)

お知らせ

▶**読者の広場**
本誌に関するご意見・ご希望などを，綴じ込みのハガキでお寄せください．読者の広場への掲載分には粗品を進呈いたします．なお，掲載に際しては表現の一部を変更させていただくことがありますので，あらかじめご了承ください．

▶**投稿歓迎**
本誌に投稿をご希望の方は，連絡先(自宅/勤務先)を明記のうえ，テーマ，内容の概要をレポート用紙1～2枚にまとめて「Interface投稿係」までご送付ください．メールでお送りいただいても結構です(送り先はsupportinter@cqpub.co.jpまで)．追って採否をお知らせいたします．なお，採用分には小社規定の原稿料をお支払いいたします．

▶**本誌掲載記事についてのご注意**

▶**コピーサービスのご案内**
本誌バックナンバーの掲載記事については，在庫(原則として24か月分)のないものに限りコピーサービスを行っています．コピー体裁は雑誌見開きの，複写機による白黒コピーです．なお，コピーの発送には多少時間がかかる場合があります．

- コピー料金(税込み)
  1ページにつき100円
- 発送手数料(判型に関わらず)
  1～10ページ：100円，11～30ページ：200円，31～50ページ：300円，51～100ページ：400円，101ページ以上：600円
- 送付金額の算出方法
  総ページ数×100円＋発送手数料
- 入金方法
  現金書留か郵便小為替による郵送
- 明記事項
  雑誌名，年月号，記事タイトル，開始ページ，総ページ数
- 宛て先
  〒170-8461　東京都豊島区巣鴨1-14-2
  CQ出版株式会社　コピーサービス係
  (TEL：03-5395-4211，FAX：03-5395-1642)

お問い合わせ先のご案内

- 在庫，バックナンバー，年間購読送付先変更に関して
  販売部：03-5395-2141
- 広告に関して
  広告部：03-5395-2133
- 雑誌本文に関して
  編集部：03-5395-2122

記事内容に関するご質問は，返信用封筒を同封して編集部宛てに郵送してくださるようお願いいたします．筆者に回送してお答えいたします．

Interface
©CQ出版(株)　2003　振替　00100-7-10665
2003年10月号　第29巻　第10号(通巻第316号)
2003年10月1日発行(毎月1日発行)
定価は裏表紙に表示してあります
発行人／増田久喜
編集人／相原　洋
編集／大野典宏　村上真紀　山口光樹　落合幸喜　小林由美子
デザイン・DTP／クニメディア株式会社
表紙デザイン／株式会社プランニング・ロケッツ
本文イラスト／森　祐子
広告／澤辺　彰　中元正夫　菅原利江
発行所／CQ出版株式会社　〒170-8461　東京都豊島区巣鴨1－14－2
電話／編集部(03)5395－2122　URL　http://www.cqpub.co.jp/interface/
広告部(03)5395－2133　インターフェース編集部へのメール
販売部(03)5395－2141　supportinter@cqpub.co.jp
CQ Publishing Co.,Ltd.／1－14－2 Sugamo, Toshima-ku, Tokyo 170-8461, Japan
印刷／クニメディア株式会社　美和印刷株式会社
製本／星野製本株式会社
日本ABC協会加盟誌
(新聞雑誌部数公査機構)　ISSN0387-9569
Printed in Japan